USENIX C++ CONFERENCE PROCEEDINGS SPRING 1991

USENIX

C++

CONFERENCE PROCEEDINGS

Washington, D.C.
April 22 - 25, 1991

Program and Table of Contents

USENIX C++ Conference

April 22 - 25, 1991
Washington, D.C.

****

Preface

I am pleased to present this collection of papers from the Third USENIX C++ Conference. The 18 papers here were selected from the 55 extended abstracts that were submitted for the program committee's consideration. All the abstracts were read and reviewed by at least three members of the committee. We were pleased at the high quality of the submissions and disappointed that we could not accept more papers.

My thanks to the members of the program committee for their hard work and careful reading and discussion of the papers. I would also like to thank Judy DesHarnais, Ellie Young, and Carolyn Carr at USENIX for all their work organizing, publicizing, and arranging the conference.

Mark Linton
Program Chair

The Interaction of Pointers to Members and Virtual Base Classes in C++

Randall Meyers
Digital Equipment Corporation
110 Spit Brook Road
ZKO 2-3/N30
Nashua, New Hampshire 03062

November 29, 1990
Revised February 26, 1991

Abstract

As originally envisioned, pointers to members in C++ had a very cheap implementation: the underlying representation of pointers to data members need be nothing more than an integer to hold the offset of the data member from the start of the class. However, this representation is insufficient for dealing with class members of virtual base classes. This paper demonstrates why this problem exists, and presents an implementation model sufficient for handling a pointer to member of a virtual base class. The paper concludes with a discussion how the semantics of the C++ language could be modified in light of the difficulties inherent in pointers to members of virtual base classes.

1 Introduction

A long recognized pitfall in computer language design is that apparently orthogonal language features sometimes interact in surprising ways. For example, one feature might prohibit common optimizations of a different feature, or one feature might require that a straightforward implementation of a different feature be made more complex.

C++ contains an instance of the second form of interaction. The existence of virtual base classes in the language complicates the underlying representation of pointers to members. There has been little public discussion of this problem. The Annotated C++ Reference Manual [1, §8.1.2c] only contains the cryptic comment that "the implementation of pointers to members depends on the way virtual base classes are represented and on the way virtual function call is implemented." A small survey of three independently written (as opposed to ports of common code) C++ compilers showed two of the compilers refused to take the pointer to member address of a virtual base class member (an unimplemented feature message was issued) while the third generated code that did not work.

This paper will explain the interaction of pointers to members and virtual base classes in six steps. First, it will describe the basic characteristics of pointers to members. Second, it will translate those characteristics into requirements on the implementation of pointers to members. Third, it will describe the Annotated C++ Reference Manual's implementation of pointers to members. Fourth, it will describe the characteristics of virtual base classes. Fifth, it will show how virtual base classes invalidate assumptions in the Annotated C++ Reference Manual's implementation of pointers to members. Sixth, it will present an alternative implementation capable of handling classes with virtual base classes.

Since any representation of pointers to members that works with virtual base classes is more expensive, and since, in general, compilers do not yet support pointers to members of virtual base classes, it is legitimate to ask how well this feature fits in the language. The final section of this paper discusses the language issues raised by pointers to members of virtual base classes.

2 Basics of Pointers to Members

Pointers to members were first covered in depth in a paper by Lippman and Stroustrup [2]. The Annotated C++ Reference Manual has revised some of the semantics of pointers to members described in that paper. When this section refers to the the original semantics, it means the semantics as portrayed in [2].

A pointer to member designates a particular member in a class independent of any particular instance of that class. The designated member can then be selected from any particular instance of the class by using the operator ".*" or from a pointer to an instance of the class using the operator "->*". Once a member is selected from a particular class instance via a pointer to member, any of the normal operations on that member are allowed.

A short example:

```
class X {public: int a, b} x1, x2;
int X::*ip;

ip = &X::b;
```

The pointer to member ip can be used to refer to the b member of any object whose type is class X. Specifically, x1.*ip refers to x1.b and x2.*ip refers to x2.b.

The type of a pointer to member includes the type of the member that can be pointed to as well as the type of the class to which the member belongs. Thus, in the above example, the type of ip is pointer to int member of class X. As you would expect, the C++ language specifies strong type checking on pointers to members.

Pointers to members can point to either data members or function members of a class. A pointer to a function member must be able to point interchangeably to a virtual or non-virtual member function.

If a pointer to member function designates a virtual function, the pointer to member function respects the delayed binding of that virtual function. For example, if pointer to member pmf designates the virtual function f, then class_ptr->*pmf() calls the same overriding instance of virtual function f as class_ptr->f() would call.

Pointers to members are first class objects in C++: there can be pointers to pointers to members, arrays of pointers to members, functions returning pointers to members, and references to pointers to members. Pointers to members may be converted to different types, assigned, passed as arguments, and compared for equality and inequality.

A predefined, standard conversion allows a pointer to member of a class to be implicitly converted to a pointer to a member of a class derived from that class provided the conversion is unambiguous. This conversion is always safe because a derived object must always contain the base class as a subobject.

The inverse of the above conversion is unsafe: a derived class may have members that do not exist in its base classes. As originally specified, explicit conversion from a pointer to member of a derived class to a pointer to member of a base class was allowed. However, the language rules have been revised to only allow such conversions for pointer to function members [1, §5.4].

Originally, a pointer to data member could be converted by explicit conversion to a `short`, or to any data type that a `short` could be converted to. This conversion is now forbidden [1, §5.4]. (`short` was the underlying representation of pointer to data members.)

A similar "representational viewing" kind of conversion was originally allowed for pointers to function members: A pointer to function member could be converted by an explicit cast to a pointer to void or to a pointer to function. If the pointer to member designated a non-virtual function, the result value of such a cast was the address of the member function. These conversions are now outlawed [1, §5.4].

As there is for normal pointers, there is a null pointer to member value. An integral constant expression with the value zero when converted to a pointer to member type is guaranteed to compare unequal to any pointer to member that actually designates a member.

The above language semantics dictate the requirements on pointers to members. The requirements for pointers to data members are distinct from those on pointers to function members, and this leads to two different representations.

A pointer to a data member has two major requirements:

- It must be able to designate a data member within a class independent of any particular instance of the class; and
- It must have a reserved value for the null pointer to member constant.

The example implementation of pointer to data member in [1, §8.1.2c] uses an integer as the underlying data type. If the pointer to member is null, then zero is stored in the integer. If the pointer to member designates a member, then one plus the offset of that member from the start of its class is stored.

A pointer to function member initially seems to have only slightly more complicated requirements:

- It must be able to designate a member function within a class independent of any particular instance of the class.
- It must be able to designate a virtual or non-virtual member function interchangeably.
- It must have a reserved value for the null pointer to member constant.

However, pointers to function members must not affect the selection of the proper overriding instance of a virtual function and must support all of the mechanics needed to call a member function. Thus, the following additional requirements need to be added to the above list:

- When selecting a member function from a particular object, if the member function is virtual, the pointer to member must inspect the virtual function table (`vtbl`) associated with the particular object in order to select the proper overriding instance of the virtual function.
- The pointer to member must allow the proper `this` pointer to be computed for the designated function.

These last two requirements, in particular, make the representation of pointers to function members dependent on the representation of `vtbl`s and the mechanism used to calculate the `this` pointer.

For example, in the model implementation outlined in [1, §10.8c], `vtbl` entries contain the address of the proper overriding instance of the member function and a delta to be added to the address of the subobject that contains the `vptr` that points to the `vtbl`. Each non-virtual base class that an object has gives rise to a different subobject in the object as a whole. The object as a whole shares its `vptr` and `vtbl` with its first non-virtual base class; the combined `vtbl` has entries for the virtual functions first declared in the base class and in the class as a whole. Each additional base class subobject has a `vptr` and `vtbl` that contains entries for the virtual functions first declared in that class.

In that scheme, invoking a virtual function requires:

1. Determining which `vtbl` contains an entry for the virtual function.
2. Determining which subobject contains the `vptr` for that `vtbl`.
3. Forming the `this` pointer based on the address of the subobject and the delta in the `vtbl`.

This causes the matching representation for pointer to virtual function member to be a triple:

1. An offset from the start of the class to the subobject containing the `vptr` (needed to find the subobject).
2. An offset from the start of the subobject to the `vptr` (needed to find the `vptr` and thus the `vtbl`).
3. An index into the `vtbl` (needed to find the address of the member function and the adjustment to the `this` pointer).

To see how sensitive pointers to member functions are to changes in `vtbl`s, consider what would happen if the `vtbl` associated the object as a whole had entries that duplicated all of the entries in all of the `vtbl`s of its base classes. Then, there would be no need to determine which `vtbl` to use or to calculate the address of the subobject containing the `vptr`. Instead, the `vptr` associated with the class as a whole could always be used, and the only piece of information needed to find the correct `vtbl` entry would be its index.

In contrast to virtual functions, invoking a non-virtual member function does not require that the `vtbl` be used, and only the address of the member function and the delta for the `this` pointer are needed.

The final representation of pointer to function members described in [1, §8.1.2c] combines both representations into a single triple:

1. An offset from the start of the class. For a virtual function, this is the offset to subobject containing the `vptr`. For a non-virtual function, this is the delta used to calculate the `this` pointer.
2. An offset or address. For a virtual function, this is the offset of the `vptr` in the subobject. For a non-virtual function, this is the address of the member function.

3. An index. For a virtual function, this is the index into the vtbl. For a non-virtual function, this index is set to a negative number to indicate that designated function is non-virtual.

[1, §8.1.2c] recommends that a value of zero in all three components indicate a null pointer to member constant. This may cause a problem for some implementations when a pointer to member designates the first virtual function in a class with no base classes and no data members. An alternative is to have a vtbl index of −1 mean that the designated function is non-virtual and a vtbl index of −2 mean that the pointer to member is null.

3 Basics of Virtual Base Classes

Virtual base classes were first described in [3].

Normally, every base class of a derived class gives rise to a separate subobject in the object as a whole. The base classes could themselves have base classes, and thus the subobjects for those base classes could contain further subobjects.

Some of these subobjects at the various levels of nesting might be of the same class type. For example:

```
class A {int i;};
class B : A {int j;};
class C : A {int k;};
class D : B, C {int l;} d;
```

Because A is a base class of both B and C, the d object contains two subobjects of type A. These objects are independent of each other, and may have different values stored in them, etc.

The fact that the subobjects are independent of each other and that the layout of the subobjects within an object or subobject is independent of the structure of any other object leads to a very important property: In any object, all subobjects and members are at fixed, known offsets. This is true independent of whether the object in question is the whole object or a subobject of a larger object.

The above "normal" case is only true for non-virtual base classes. In contrast, virtual base classes do not give rise to multiple, separate subobjects if they are used multiple times as direct or indirect base classes. Instead, a single instance of the virtual base class subobject is shared by all of the subobjects derived from it in the object as a whole. For example:

```
class A2 {int i;};
class B2 : virtual A2 {int j;};
class C2 : virtual A2 {int k;};
class D2 : B2, C2 {int l;} d2;
```

Because A2 is a virtual base class of B2 and C2, the d2 object contains only one subobject of type A2. The B2 and C2 subobjects both share this one instance of A2.

With virtual base classes, subobjects within an object or subobject are no longer independent of each other since they might share (or be) a common virtual base class. Because of this, the layout of an object might change if it is a subobject within a more derived object instead of being the object as a whole.

Consider the previous example. Within an object of type B2, the offset of the subobject for the virtual base class A2 might be 8 bytes. Likewise, within an object of type C2, the offset of the A2 subobject might be 8 bytes. However, within an object of type D2, at least one of these offsets must change since there can only be one instance of A2 within the D2 object, and it can not appear at two different offsets (the two offsets being the offset of B2 + 8 and the offset of C2 + 8).

The lesson of the above example is that a C++ compiler can not assign a constant offset to a virtual base class subobject within a class object unless it knows about all of the classes that will be derived from the class of the object: any further derivation could introduce additional uses of virtual base classes, and thus introduce additional dependencies on a compatible layout of the object. Since separate compilation prevents C++ compilers from knowing all of the derivations from a class, the offset of virtual base class subobjects is not fixed: The offset to a virtual base class subobject is likely to differ when the object containing the virtual base class is itself a subobject instead of an object as a whole.

The above has significant consequences for the language: when dealing with a pointer to a class object, a compiler does not know whether the pointer points to a whole object or whether it points to a subobject within a derived class. This means a compiler cannot know at compile-time the offset of any member located in a virtual base class of the object that the pointer points to. Thus, the compiler must generate code to determine the offset at runtime.

The implementation of virtual base classes uses indirection to allow a compiler to find the offsets of the virtual base class members. A word is set aside at a fixed location within any object containing a virtual base class. This word contains the address of the virtual base class subobject. The virtual base class subobjects are then allocated at the end of the object after all of the non-virtual subobjects and members. Thus, all of the non-virtual members are at known, fixed offsets for all instances of a class (subobject or object as a whole), and all of the virtual base class members can be reached indirectly based on bookkeeping information stored at known offsets.

4 A Representation for Pointers to Members of Virtual Base Classes

The implementation model of pointers to members in the Annotated C++ Reference Manual assumes that all members of a class are at fixed, known offsets that do not change whether the class is a whole object or is a subobject within a further derived object.

The representation of pointers to data members assumes this by attempting to store the offset of the member (plus one). The representation of pointers to function members assumes this by requiring the offset from the start of the object to the subobject that contains the vptr.

As Section 3 shows, the offset of any virtual base class member in a class is not independent of the instance of the class since one instance may be a whole object while another may be a subobject. This violates the assumption that the offset is fixed, and thus prevents the implementation model in [1] from allowing a pointer to member to designate a member independent of any particular instance of a class.

Even though the offset of a virtual base class member is not invariant, it is possible to find the virtual base class subobject that the member is in at runtime by following the pointer to the virtual subobject. Once the proper subobject is located, the member is at a fixed offset from the start of the subobject. The algorithm for finding a member of a virtual base class is:

1. Find the pointer to the virtual base class. This pointer is a fixed, known offset.
2. Indirect through that pointer to find the virtual base class subobject.
3. The desired member is now at a fixed, known offset in that subobject.

Since virtual base classes may themselves have virtual base classes, steps 1 and 2 may have to be repeated for each level of nesting.

A representation for pointers to data members that allows the above procedure to be carried out is a counted vector of offsets. The first element of the vector is a count of the number of elements that follow. Each of the following elements is a successive offset into the subobject at the current level of indirection. Each of these offsets except for the last is the offset to a virtual base class pointer; the last of these offsets is the offset of the desired member.

The following C function calculates the address of a member given the address of a class instance and a pointer to member in the above format:

```
/* Return &object.*ptr_to_mem */
char *member_address(
    char *object,                       /* address of object */
    unsigned long ptr_to_mem[])         /* pointer to member */
{
    unsigned long i;
    char *subobj;

    /* A count of zero means pointer to member is null */
    if (ptr_to_mem[0] == 0)
        return NULL;

    subobj = object;

    for (i = 1; i < ptr_to_mem[0]; ++i) {
        /* Set subobj to address of virtual subobject */
        subobj = *(char **) (subobj + ptr_to_mem[i]);
    }

    /* Return address of member within subobject */
    return subobj + ptr_to_mem[i];
}
```

Note that the null pointer to member constant is represented by an element count of zero.

A different number of elements in the vector are used depending on the particular member whose pointer to member address is being represented:

- If the member is not in located in any virtual subobject, then two elements are used: The first element (the count) is set to one and the offset of the member is stored in the second element.
- If the member is located in a virtual base class not nested in any other virtual base class, then three elements are used: The first element (the count) is set to two, the

second element is set to the offset of the virtual base class pointer, and the third element is the offset of the desired member.

- If the member is in a virtual base class of a virtual base class, then four elements are used: the count is set to three, the second and third elements are offsets of virtual base class pointers, and the fourth element is the offset of the desired member.

Although the number of elements used varies with the particular member whose pointer to member address is stored, the allocated size of the vector does not. Since a pointer to member must be able to designate any member, enough storage must be allocated to allow it to handle the worst case for the class type whose members it designates. The worst case is determined by the maximum depth of nesting of virtual base classes for the class type. If a class has `n` levels of nesting of virtual base classes, then pointers to members for that class must have `n+2` elements. Thus, a class with exactly one virtual base class would have three element pointers to members.

Pointers to function members can be extended to work with member functions of virtual base classes by replacing the offset to the subobject containing the `vptr` by a counted vector to locate the subobject. The resulting representation is:

1. A counted vector suitable for representing a pointer to data member of the class. For a virtual function, calling the previously described `member_address()` function on this vector returns the address of the subobject containing the `vptr`. For a non-virtual function, calling `member_address()` returns the `this` pointer needed for the member function call.

2. An offset or address. For a virtual function, this is the offset of the `vptr` in the subobject. For a non-virtual function, this is the address of the member function.

3. An index. For a virtual function, this is the index into the `vtbl`. For a non-virtual function, this index is set to a negative number to indicate that designated function is non-virtual.

The null pointer to member constant can be represented by an element count of zero in the counted vector.

5 Language Issues

By necessity, a representation for pointers to members that can handle members of virtual base classes must be able to handle the indirection inherent in virtual base classes. This makes the implementation of pointers to members more expensive than originally envisioned. Since support for pointers to members of virtual base classes is weak in several existing compilers, it is legitimate to ask whether the C++ language should be changed in this area.

Three alternatives present themselves. In order of decreasing reasonableness, they are:

1. Keep the language as it is (without any special rules about pointers to members and virtual base classes), and add the needed support to compilers.

2. Outlaw taking the pointer to member address of virtual base class members. This is the *status quo* for compilers that issue a "not implemented" message.

3. Modify the language to differentiate between pointers to non-virtual base class members and pointers to virtual base class members.

The primary argument against alternative 1 is that adding the support to compilers makes pointers to members more expensive. However, the expense is not as large as it first seems: The counted vector representation need only be used for pointers to members of classes that have (directly or indirectly) virtual base classes. If a class does not have any virtual base classes, then the representation for pointers to members outlined in [1] can be used. Thus, the expense is only incurred when the feature is used.

The problem with alternative 2 is that it restricts the language in a way that may seem arbitrary to programmers.

Alternative 3 splits the different between alternatives 2 and 3, and inherits some of the drawbacks of both. The idea behind alternative 3 is that part of the type of a pointer to member is whether it can designate a member in a virtual class. Thus, a pointer to an int member of class C would be prohibited from designating a member in a virtual base class of C, but a pointer to a possibly virtual int member of class C could designate a virtual or non-virtual base class member. A declaration of such a pointer might look like:

```
virtual int C::*memptr;
```

This puts control over the representation of a pointer to member in the hands of the programmer. The programmer can choose the less expensive representation for a pointer to member for a class even though the class has virtual base classes, if the programmer is willing to forgo the ability to designate members of virtual base classes. However, this control seems a small payoff when weighed against the complexity added to the language rules. This can also lead to a great deal of confusion: the presence or absence of the virtual keyword when declaring a pointer to function member has nothing to do with whether the designated function can be a virtual function or not.

In order to keep the complexity of the programmer's view of C++ to a minimum, the best choice just seems to be to make pointers to members of virtual base classes work.

Adopting a representation for pointers to members that solves the virtual base class problem has implications on the operations on pointers to members. Particularly affected are conversions of pointers to members.

Converting a pointer to member to a pointer to member of a derived class has two cases: The derived class has the current class as a non-virtual base class or as a virtual base class. In the first case, the conversion is performed by adjusting the offset stored in element 1 of the counted vector by the offset of the base class in the derived class. In the second case, the conversion causes the counted vector to grow by another element: all of the elements of the vector with index greater than 0 are shifted into the element with the next higher index. The offset from the start of the derived class to the pointer to the virtual base class is then stored in element 1. Element 0 is incremented by one. Note that the conversion in either case will cause the amount of storage allocated for the counted vector to increase if the derived class has a deeper nesting of virtual base classes.

The inverse of the above conversion is only allowed for pointers to member functions. That conversion can be carried out by inverting the operations above on the counted vector portion of the pointer to function member.

Although the conversion from pointer to data member to integer is now forbidden [1, §5.4], some compilers continue to allow it. The conversion can be used to obtain the offset of the designated member, a value that may be useful when interfacing to non-C++ code.

However, if the designated member is located in a virtual base class, this conversion has no reasonable meaning. A pointer to member that is in a virtual base class is a series of offsets and indirections that cannot be boiled down to an offset in the absence of a particular instance of the class. Compilers should probably drop the conversion entirely, or limit it to pointers to members of classes that have no virtual base classes.

The similar obsolete conversion from pointer to function member to pointer to void or pointer to function can still be done if the member function designated by the pointer to member is not a virtual function.

The new representation also has ramifications on the equality operators. Two non-null pointers to members should compare equal to each other if and only if they both designate the same member. However, with virtual base classes, two pointers to members might designate the same member in a virtual base class, but because the pointers to members use different virtual base class pointers to reach the virtual base class subobject, the pointers to members would have different values. This situation might arise in the following program:

```
class A {public: int i;};
class B : public virtual A {};
class C : public virtual A {};
class D : public B, public C {} d;

int B::*bptr = &B::i;
int C::*cptr = &C::i;
int D::*d1ptr = bptr;
int D::*d2ptr = cptr;
```

Because `d1ptr` probably follows the `B` subobject's pointer to virtual class `A` and because `d2ptr` probably follows the `C` subobject's pointer, `d1ptr` and `d2ptr` will structurally compare unequal, even though `&d.*d1ptr` compares equal to `&d.*d2ptr`. Solving this problem requires that pointers to members not be compared by comparing the values stored within them directly; instead the ultimate offset within a class that those values specify must be determined and compared.

The last effect of the more general representation of pointers to members is on the `vtbl`. For aesthetic reasons, [1, §8.1.2c, §10.8c] recommends that `vtbl` entries have the same representation as pointers to function members. This advice is reasonable when the representation of pointers to function members only has one additional word above the requirements of `vtbl` entries. However, it is unwise to adopt a `vtbl` entry representation that has all of the overhead of counted vectors.

6 Conclusions

The example representation for pointers to members described in the Annotated C++ Reference Manual is insufficient to handle members of virtual base classes. A sufficient representation must allow for the possible multiple levels of indirection required by virtual base classes.

One representation that is sufficient for pointers to members of virtual base classes is the counted vector of offsets. This representation is flexible enough to designate any member of a class at any level of indirection. The counted vector is more expensive than the representation in the Annotated C++ Reference Manual, but it only is needed when a class actually has virtual base classes.

Pointer to member of virtual base classes is poorly supported in existing C++ compilers, but there is little reason for this situation to continue. Although the language becomes more complicated to implement if compilers support pointers to members of virtual base classes, the rules in the language are somewhat simpler for programmers.

The new representation for pointers to members complicates conversion operations on pointers to members, but all of the conversions allowed by the Annotated C++ Reference Manual can be done. However, the obsolete conversion from a pointer to data member to an integer is no longer reasonable.

Pointers to members of virtual base classes raises an issue with the equality of pointers to members: two pointers to members that designate the same member in a virtual base class may not compare equal.

7 Acknowledgments

I would like to thank my reviewers at Digital: Henryk Halicki, Aron Insinga, Bill McKeeman, and Walter van Roggen.

References

[1] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[2] S. B. Lippman and B. Stroustrup. "Pointers to Class Members in C++." *Proceedings, USENIX C++ Conference*, 1988.

[3] B. Stroustrup. "Multiple Inheritance for C++." *Proceedings, EUUG Spring '87 Conference*, 1987.

Problems With Non-invasive Inheritance In C++

Martin D. Carroll
email: carroll@mozart.att.com
AT&T Bell Laboratories
184 Libery Corner Road
Warren, New Jersey 07059-0908

Inheritance is *non-invasive* if it can be done without touching the source code of the base class. Because programmers regularly make all kinds of simplifying assumptions when they write code, completely non-invasive inheritance is currently rarely possible. The purpose of this paper is to enumerate, via a running example, some of the common ways in which base classes implemented using common programming techniques may have to be changed by programmers who inherit from them. The example will be a pair of tree classes.

1 Introduction

Programmers regularly make all sorts of simplifying assumptions in their code. For example, the programmer who writes

```
int max(int i, int j) {
    return (i>j? i : j);
}
```

assumes that `max` needs only to find the maximum of two integers, never two floats.

It is impossible to program with no simplifying assumptions. Unfortunately, each simplifying assumption makes it difficult to extend the code in ways that violate that assumption. In the above case, the only way to extend `max` to floats is to replicate the code, changing type declarations:[1]

```
float max(float i, float j) {
    return (i>j? i : j);
}
```

Programming languages can make it easier or harder to write *extensible code.* It is easier to write extensible code in C++ than in C, since C++ has inheritance. For example, in C++ we can extend a previously implemented class Foobar into a class Blue_foobar via inheritance:

```
class Foobar { ... };
class Blue_foobar : public Foobar { ...};
```

[1] Or use templates.

In the best of all possible worlds, we would like to be able to do this *non-invasively*, that is, without touching the source code for Foobar. Unfortunately, completely non-invasive inheritance is currently rarely possible. Unless the inheritance is of a very specific kind explicitly intended by Foobar, the implementer of Blue_foobar has a high probability of having to go back into the class Foobar and change the source code in ways that remove certain simplifying assumptions Blue_foobar does not satisfy.

This is not a fault of the way inheritance is implemented in C++. It is instead the fault of the way programmers currently program. Programmers are not yet *used* to implementing their classes extensibly. In order to understand how to implement extensibly, it is necessary first to understand the kinds of problems programmers can encounter when they try to extend classes implemented by others.

Thus, the purpose of this paper is to enumerate, via an example, some of the common ways in which base classes implemented using common programming techniques may have to be changed by programmers who inherit from them. The example will be a pair of tree classes. We will begin by implementing a binary search tree class; we will then try to extend that to a red-black tree [1] class. (The reader interested in the complete definition and algorithms pertaining to red-black trees is referred to [1].)

A note on polymorphism A perennial question that arises when designing C++ classes is the question "Should this member function be declared virtual?" This question is actually much more complex than an uninitiated programmer would imagine; in particular, it is definitely *not* equivalent to the question "Will this member function ever be overridden in any derived class?"

One reason (but not the only reason) for making member functions virtual is so that clients can use base classes polymorphically. For example, given the following class definitions:

```
class B {
public:
    virtual foo();
};
class D : public B {
public:
    foo();
};
```

the client can write a polymorphic function taking a B and be guaranteed that it works correctly:

```
void call_foo(B& b) {
    b.foo();
}
main() {
        B b;
        call_foo(b);  // calls B::foo
        D d;
        call_foo(d);  // calls D::foo
    }
```

This would seem to argue that all public member functions overridden by derived classes should be made virtual.

Virtual functions, however, are not without cost. They require a few extra instructions to invoke (compared to their nonvirtual counterparts), and, more important, they cannot be inlined.[2] The class designer should impose this cost on the client only if the client is getting something for it. In particular, if the client has no need to write a polymorphic function, then there is no need (at least not for this reason) to make the member functions virtual.

We call the assumption that the client has no need to write a polymorphic function such as above the "*No polymorphic usage*" assumption.

In the remainder of this paper, we will assume "No polymorphic usage." Dropping this assumption would not invalidate the discussion, it would simply change some of the code we present. We chose to assume "No polymorphic usage" to demonstrate that even with "No polymorphic usage," some member functions may *still* have to be virtual.

But we're getting ahead of ourselves...

2 The binary search tree class

In this section, we will implement a binary search tree class. Since this section merely lays the groundwork for the next section, we will proceed rather quickly.

We will model the nodes in a binary search tree by a class which we will call Bsnode. A Bsnode has a key of some user-selectable type, and pointers to its left child, right child, and parent. It is also a friend of the binary search tree class, which we will call Bstree:[3]

```
template <class T> class Bstree;

template <class T> class Bsnode {
    friend Bstree<T>;
    Bsnode<T> *left, *right, *p;
    T key;
    Bsnode(const T& key_) : key(key_), left(0), right(0), p(0) {}
    ~Bsnode() {}
};
```

[2]At least not without a much smarter runtime system than is currently found in implementations of C++.
[3]Readers not familiar with C++ template syntax should consult [2].

Notice we've made the constructor private. This is so no one but Bstree will be able to construct instances of Bsnode.[4]

Binary search trees have three operations: one to insert a given key into the tree, another to delete a given key (if present), and a third to test whether a given key is in the tree. It is handy also to have an output operation.[5] To keep this example small, we will show only the insertion and output operations. Here is the complete interface, including some private member functions we will need:

```
template <class T> class Bstree {
    int doinsert(const T &key, Bsnode<T>* px, Bsnode<T>*& x);
    void print(ostream &os, const Bsnode<T>* x, int depth) const;
    void delete_subtree(Bsnode<T>* x);
    Bsnode<T>* root;
public:
    Bstree() : root(0) {}
    ~Bstree() {
        delete_subtree(root);
        root = 0;
    }
    // returns 1 if the key was not present
    int insert(const T &key) { return doinsert(key, 0, root); }
    void print(ostream &os) const { print(os, root, 0); os << endl; }
};
```

The following is just the standard binary search tree recursive insertion algorithm. The parameter **x** is the node whose key we are currently examining, **px** is the intended parent of the inserted node, and **key** is the key we are inserting:

```
template <class T>
int
Bstree<T>::doinsert(const T &key, Bsnode<T>* px, Bsnode<T>*& x) {
    if (x == 0) {
        x = new Bsnode<T>(key);
        x->p = px;
        return 1;
    }
    else if (key < x->key)
        return doinsert(key, x, x->left);
    else if (key > x->key)
        return doinsert(key, x, x->right);
    else
```

[4]We could have made it protected and still achieve this protection. However, I am being true to my story: when I first wrote this code, I made it private.

[5]Since the version of C++ I am working with only supports class templates and not function templates, I will make the output operation a member of class Bstree, rather than follow the normal approach and overload the global left-shift operator.

```
        return 0;
}
```

To delete a subtree rooted at a given node, we recursively delete the left and right subtrees of that node, and then delete the node itself:

```
template <class T>
void
Bstree<T>::delete_subtree(Bsnode<T>* x) {
    if (x != 0) {
        delete_subtree(x->left);
        delete_subtree(x->right);
        delete x;
    }
}
```

Output is simply a depth-first search, printing keys as we go along:

```
template <class T>
void
Bstree<T>::print(ostream &os, const Bsnode<T>* x, int depth) const {
    for (int i=0; i<depth; i++)
        os << "  ";
    if (x == 0)
        os << '-' << endl;
    else {
        os << x->key << endl;
        print(os, x->right, depth+1);
        print(os, x->left, depth+1);
    }
}
```

Notice we keep track of the current depth in order to get the indentation right.

Here is a sample program:

```
main() {
    Bstree<int> t;
    t.insert(6);
    t.insert(2);
    t.insert(4);
    t.insert(7);
    t.insert(4);
    t.insert(5);
    t.print(cout);
}
```

This program prints out the following:

```
6
  7
    -
    -
  2
    4
      5
        -
        -
      -
    -
```

Now let's try to extend this class.

3 The red-black tree class

A red-black tree is simply a binary search tree in which every node additionally has a *color*, either red or black, and the tree satisfies certain balance criteria[6] An insertion or deletion in a red-black tree may upset the balance criteria; if this occurs, the tree is rebalanced in such a way that symmetric ordering is preserved, and the balance criteria are restored.

First we model red-black tree nodes as binary search tree nodes with an additional color:

```
enum Color { red, black };
template <class T> class Rbtree;
template <class T> class Rbnode : public Bsnode<T> {
    friend Rbtree<T>;
    Color color;
    Rbnode(const T& key) : Bsnode<T>(key), color(red) {}
    ~Rbnode() {}
};
```

Notice that red-black nodes are initialized red. Later algorithms will use this fact.

Next we model red-black trees themselves. A red-black tree is simply a binary search tree with a different (extended) insertion operation, and a rebalancing operation:

```
template <class T> class Rbtree : public Bstree<T> {
    void rebalance(Rbnode<T>* x);
    void left_rotate(Rbnode<T>* x);
    void right_rotate(Rbnode<T>* y);
public:
    int insert(const T& key);
};
```

[6]Specifically, the length of the longest external path cannot be more than twice as large as the length of the shortest external path.

The rotation operations [1] will be used by `rebalance`. We will implement them first. Here is the algorithm for left rotation:

```
template <class T>
void
Rbtree<T>::left_rotate(Rbnode<T>* x) {
    Rbnode<T>* y = (Rbnode<T>*)x->right;
    x->right = y->left;
    if (y->left != 0)
        y->left->p = x;
    y->p = x->p;
    if (x->p == 0)
        root = y;
    else {
        if (x == x->p->left)
            x->p->left = y;
        else
            x->p->right = y;
    }
    y->left = x;
    x->p = y;
}
```

Unfortunately, when we try to compile this, we get the following error:[7]

```
Rbnode<int>::Rbnode<int>() cannot access Bsnode<int>::Bsnode<int>(): private member
```

This is because we tried to inherit Rbnode from a class (Bsnode) whose constructor is private. (Remember, we made it private to prevent outsiders from creating instances of it.) Now that we have to inherit from Bsnode, we must make its constructors protected. We should also make its other members protected as well, since we will have occasion to use them in Rbnode:

```
template <class T> class Bsnode {
protected:
    // same as before
};
```

Base class change #1: Some private members must be made protected.

Since we had to make the members of Bsnode protected, we should probably do the same for Rbnode:

[7]In the following, when we say "try to compile this," we mean "try to compile this, plus a declaration of the form "`Rbnode<int> r`". In my C++ translator, template errors are revealed only when we try to expand the template.

```
template <class T> class Rbnode {
protected:
    // same as before
};
```

Compiling the new program still shows a protection problem:

```
Rbtree<int>::left_rotate() cannot access Bstree<int>::root: private member
```

This is the same problem revisited: the `root` member of Bstree must be made protected so the derived class Rbtree can manipulate it. As for the the other members of Bstree, we shall leave them private, since derived classes will have no need to access them, and in general we prefer to keep protections as restrictive as possible. Here is the modified definition of Bstree:

```
template <class T> class Bstree {
protected:
    Bsnode<T>* root;
    // rest same as before
};
```

Surprisingly enough, the resulting program still has a protection problem:

```
Rbtree<int>::left_rotate() cannot access Bsnode<int>::p: protected member
```

The offending line is the following:

```
y->left->p = x;
```

The problem is rather subtle: although Rbnode is a friend of Rbtree, the static type of `y->left` is `Bsnode<T>*`, not `Rbnode<T>*`. Rbtree does not have access to the members of Bsnode. A possible solution is to make Rbtree a friend of Bsnode, but that would require a base class change (adding the friend declaration to Bsnode), and in any case it misses the point: in this particular case we know that `y->left` is actually pointing to an Rbnode, not a Bsnode. Rbtree *does* have access to the members of Rbnode. Thus, a better solution would be to insert the appropriate cast:

```
((Rbnode<T>*)(y->left))->p
```

However, this is ugly, and furthermore we would have to do it in many places throughout the code. The best solution is to define *cast wrappers* in the class Rbnode:

```
template <class T> class Rbnode : public Bsnode<T> {
    Rbnode<T>* Left() const { return (Rbnode<T>*)left; }
    // similarly for right and p
};
```

These wrappers simply reinterpret the types of the inherited pointers to be whatever they actually are in the derived class. Now we can rewrite the function `left_rotate` as follows:

```
template <class T>
void
Rbtree<T>::left_rotate(Rbnode<T>* x) {
    Rbnode<T>* y = x->Right();  // no more cast needed
    x->right = y->Left();
    if (y->Left() != 0)
        y->Left()->p = x;
    y->p = x->P();
    if (x->P() == 0)
        root = y;
    else {
        if (x == x->P()->Left())
            x->P()->left = y;
        else
            x->P()->right = y;
    }
    y->left = x;
    x->p = y;
}
```

This is neater than proliferating casts all over the code. It is also safer: in general, casts should be localized to as few places in one's program as possible. Notice, by the way, that when *assigning* to a "wrapped" inherited pointer, we do *not* use the cast. A statement such as

```
x->P() = y;
```

would be illegal, since the left hand side is not an lvalue. The correct statement

```
x->p = y;
```

is both legal and safe, since we are assigned a pointer-to-derived-class to a pointer-to-base-class.

Finally, as long as we're wrapping, let's do it for Rbtree too:

```
template <class T> class Rbtree : public Bstree<T> {
    Rbnode<T>* Root() const { return (Rbnode<T>*)root; }
    // rest same as before
};
```

The code for `right_rotate` looks similar to `left_rotate`, and we omit it. We also omit the code for `rebalance`.

Now let's try to write `insert`. To insert in a red-black tree, we simply insert in the underlying binary search tree, and then, if the value was not already present, we rebalance on the new node:

```
template <class T>
int
Rbtree<T>::insert(const T& key) {
    if (Bstree<T>::insert(key)) {
        rebalance(???);
        return 1;
    }
    return 0;
}
```

Notice the problem: when we go to rebalance, we have no handle on the new node! Although the function `Bstree::doinsert` had a pointer to the new node, that information was "lost" by the time `Bstree::insert` returned. We must go back and change the base class implementation to preserve that information and make it accessible to the derived class. Here is the new implementation of `Bstree`:

```
template <class T> class Bstree {
protected:
    Bsnode<T>* the_node;
    // rest same as before
};

template <class T>
int
Bstree<T>::doinsert(const T &key, Bsnode<T>* px, Bsnode<T>*& x) {
    if (x == 0) {
        x = new Bsnode<T>(key);
        x->p = px;
        the_node = x; // ADDED THIS
        return 1;
    }
    else {
        if (key < x->key)
            return doinsert(key, x, x->left);
        else if (key > x->key)
            return doinsert(key, x, x->right);
        else {
            the_node = x;  // ADDED THIS
            return 0;
        }
    }
}
```

Base class change #2: Some information lost in the base class must be restored for the derived class.

Before we can implement `Rbtree::insert` cleanly, we need a cast wrapper for `the_node`:

```
template <class T> class Rbtree : public Bstree<T> {
    Rbnode<T>* The_node() const { return (Rbnode<T>*)the_node; }
    // rest same as before
};
```

Now we can implement `Rbtree::insert` as follows:

```
template <class T>
int
Rbtree<T>::insert(const T& key) {
    if (Bstree<T>::insert(key)) {
        rebalance(The_node());
        return 1;
    }
    return 0;
}
```

Our program finally compiles. Unfortunately, when we run it, we get:

```
Memory fault(coredump)
```

What went wrong? A bit of inspection reveals the answer: in the implementation of the base class function `Bstree::doinsert`, when we determine that we need to insert a new node into the tree, we do the following:

```
x = new Bsnode<T>(key);
```

For the class Bstree, this is perfectly correct. Unfortunately, for the class Rbtree (which inherits the implementation of `doinsert`), it is not correct. When we insert a new node into an Rbtree, we need to construct an Rbnode, not a Bsnode. The above line, in order to be correct for *both* Bstree and Rbtree, should really read:

```
x = new <type of node used by this tree>(key);
```

That is, we need a *virtual constructor* [3,4]. Since C++ does not allow constructors to be declared virtual, (it is not clear what the semantics of that would be in any case), we must fake one via a virtual function. First we change the base class as follows:

```
template <class T> class Bstree {
    virtual Bsnode<T>* newnode(const T& key) const { return new Bsnode<T>(key); }
    // rest same as before
};
```

This function delivers a new instance of whatever type of node is associated with this tree. In `Bstree::doinsert` we replace the line

```
    x = new Bsnode<T>(key);
```

with

```
    x = newnode(key);
```

Base class change #3: Base class must simulate virtual constructors.

In Rbtree, override newnode to construct an Rbnode instead of a Bsnode:

```
template <class T> class Rbtree : public Bstree<T> {
    Bsnode<T>* newnode(const T& key) const { return new Rbnode<T>(key); }
    // rest same as before
};
```

Since we have just made a constructor call for Bsnode "virtual," we must now make the destructor for Bsnode virtual. To see why, consider the implementation of `Bstree::delete_subtree`:

```
template <class T>
void
Bstree<T>::delete_subtree(Bsnode<T>* x) {
    if (x != 0) {
        delete_subtree(x->left);
        delete_subtree(x->right);
        delete x;
    }
}
```

Notice the line "`delete x`". Since the constructor call for Bsnode has been made "virtual," it is no longer the case that the object pointed to by `x` is always a Bsnode. When `delete_subtree` is called in the derived class, `x` will actually point to an Rbnode. In order for the right thing to be destroyed, we must make the Bsnode destructor virtual:

```
template <class T> class Bsnode {
protected:
    virtual ~Bsnode() {}  // ADDED ``virtual''
    // rest same as before
};
```

Base class change #4: Some base class functions must be made virtual.

This problem may be rather surprising, in light of our "No polymorphic usage" assumption stated earlier. *Even if we assume no polymorphism on the part of the client*, we may still have to make certain member functions virtual.

The generalized static binding problem The above two base class changes are both symptoms of a general problem, which I call the *generalized static binding* problem. Normally the term "static binding" is used to refer to the binding between a function name and a particular instance of that function. For instance, in the call

```
f(X& x) {
    x.f();
}
```

if `f` is a non-virtual member function of `X`, then the function name "f" in this call is said to be statically bound to the definition of `f` in `X`.

There are many other possible kinds of static binding. One of the most common kinds, which we make whenever we program, is static binding between the *intent* of a piece of code and the actual code used to implement it. I call this *implementation binding.* For example, in the virtual constructor problem above, there was an implementation binding between the intended type "type of node used by this tree" and the actual type Bsnode. In the case of the virtual destructor problem, there was an implementation binding between the intended "destructor for node x," and the actual destructor `Bsnode::~Bsnode`.

Just as it is not easy to see function name static bindings in source code (since the "virtual" keyword, if present, is located elsewhere in the program), it is even harder to see implementation bindings, since the intent of a piece of code is not expressed in the language. However, when we inherit from a class, we inherit its implementation bindings just as surely as we inherit its function name bindings. If *any* of these bindings are wrong for the derived class, then we must "invade" the base class to loosen them.

Are there any other inherited bindings in our code which should be loosened for the derived classes? Execution of the program so far shows that there is one more. Here is the sample program of the previous section, change to use an Rbtree instead of a Bstree:

```
main() {
    Rbtree<int> t;
    t.insert(6);
    // ...
    t.insert(5);
    t.print(cout);
}
```

Here is the output generated by our program so far:

```
4
  6
    7
      -
      -
    5
      -
```

```
        -
    2
      -
      -
```

Notice that the colors of the nodes are missing. Here are the relevant portions of the code, which reveal why:

```
template <class T>
void
Bstree<T>::print(ostream &os, const Bsnode<T>* x, int depth) const {
    // ...
    os << x->key << endl;
    // ...
}
template <class T> class Rbtree : public Bstree<T> { ... };
```

Notice that what print really wants to do is "output node x." However, when Rbtree inherits `print` from Bstree, it inherits the implementation binding of this to the specific piece of code

```
os << x->key;
```

Although this is perfectly correct for Bsnode, for Rbnode it should be

```
os << x->key << " (" << x->color << ")";
```

The way to loosen this binding for the derived class is to replace the static binding in `print` with a call to a virtual function:

```
template <class T> class Bsnode {
    virtual void print(ostream &os) const { os << key; }
    // rest same as before
};

template <class T>
void
Bstree<T>::print(ostream &os, const Bsnode<T>* x, int depth) const {
    // ...
    x->print(os);
    os << endl;
    // ...
}
```

Base class change #5: Some base class hard code may have to be delegated to virtual functions.

In Rbnode, we override the `print` function:

```
template <class T> class Rbnode : public Bsnode<T> {
    void print(ostream &os) const { os << key << " (" << color << ")"; }
    // rest same as before
};
```

Now the program works as expected:

```
4 (black)
  6 (black)
    7 (red)
      -
      -
    5 (red)
      -
      -
  2 (black)
    -
    -
```

4 Conclusion

Inheritance is *non-invasive* if it can be done without touching the source code of the base class. Because programmers regularly make many simplifying assumptions when they write code, completely non-invasive inheritance is rarely possible. In this paper, we showed the typical ways in which base classes — perfectly good, perfectly well-designed base classes — in C++ may have to be changed by programmers who inherit from them.

Acknowledgements Thanks to Mike Vilot, one of whose comments at C++ At Work '90 made me see how to explicate this paper. Thanks also to Steve Buroff for his always enlightening discussions.

References

[1] T. Cormen, C. Leiserson, and R. Rivest. 1990. *Introduction to Algorithms.* The MIT Press.

[2] M. Ellis and B. Stroustrup. 1990. *The Annotated C++ Reference Manual.* AT&T Bell Laboratories.

[3] D. Jordan. 1989. Class derivation and emulation of virtual constructors. *The C++ Report*, 1(8), September.

[4] M. Carroll and M. Ellis. 1991. Virtual constructors in C++. *The C++ Journal.* forthcoming.

Automatic Detection of C++ Programming Errors: Initial Thoughts on a lint++

Scott Meyers and Moises Lejter

sdm@cs.brown.edu and mlm@cs.brown.edu
Department of Computer Science
Brown University, Box 1910
Providence, RI 02912

Abstract

In this paper we argue that there is sufficient experience with C++ to justify the development of a tool that examines C++ programs for the presence of likely programmer errors, and we describe a number of common mistakes that could be detected by such a tool. We show that such a tool would be both straightforward to develop and efficient to apply. We also discuss how such a tool could be extended to detect violations of design constraints expressed in some as-yet-to-be-developed C++ metalanguage.

Introduction

It is a fact of life that programmers make mistakes when writing programs. The kinds of mistakes that interest us are not those that are errors (i.e., violations of the rules of the language), but are instead legal constructs that mean something other than what the programmer likely intends them to mean. Some of the mistakes of this kind are straightforward enough that it is possible to write a program to detect them, and some are common enough that it is worthwhile to do so.

C programmers often rely on lint to ensure that their software is free of common errors of this kind. In some sense, lint is an distillation of the collective historical experience of the C programming community, a terse summary of the lessons learned by many people from many programs over a long period of time. In this paper, we propose that there is now sufficient experience with C++ to initiate the development of a C++ counterpart to lint, a lint++.

Since C++ is based on C, a straightforward approach would be to base a lint++ on lint. Our experience in working with C++ and in teaching the language to scores of professional C programmers, however, indicates that the kinds of errors made in C++ are fundamentally different from those made in C, and are a direct outgrowth of the new features offered by C++. In this paper, we identify a number of common C++ programming errors, and we describe how a lint++ could detect them.

In compiling the list that follows, our goal has been to identify constructs that are "almost always wrong." The "almost always" part of this philosophy makes explicit our recognition that there are some legitimate uses for these constructs, but such uses are comparatively rare. The "wrong" part means that the construct fails to do what the programmer intends it to do, the construct violates accepted language convention, or the construct violates accepted software engineering practice. By "wrong" we do *not* mean "bad style."

Stylistic conventions vary radically from place to place, and we have no illusions that we might be able to foist our stylistic preferences on the C++ community by codifying them in the rules that follow.

Anything that is almost always wrong is occasionally right. As a practical matter, this means that any lint++ must provide programmers with some mechanism for suppressing warnings where the programmers find them to be inappropriate. The details of such a mechanism do not concern us here, but the scope of such suppression will probably need to include at least statements, files, and runs.

Candidate Errors

We propose that a lint++ should detect at least the following conditions:

1. **Failing to match constructor calls to new with destructor calls to delete.**

   A common source of memory leaks is to allocate memory in a constructor without deallocating it in the destructor. The discrepancy often arises when a new pointer member is added to a class. The pointer must typically be initialized or assigned in each class constructor as well as in the class assignment operator, and after making the appropriate changes to these member functions, it is easy (and common) to forget to update the class destructor to delete the memory referenced by the pointer.

   A more ambitious lint++ might differentiate between simple pointers and pointers to arrays, issuing a warning if a pointer to an array were not deleted using the array deletion syntax.

2. **Declaring a non-virtual destructor.**

   The following code skeleton is very common:

   ```
   Base *bp = new Derived;
   ...
   delete bp;
   ```

   Unfortunately, the destructor for class Derived is never called unless the destructor for class Base is declared virtual, a fact overlooked by almost all new C++ programmers. Fortunately, detection of this error is simple. A more sophisticated lint++ might want to issue a warning only if the class defines at least one virtual function. In fact, the justification for destructors not being virtual by default is that it is sometimes essential to create classes requiring no virtual table[ES90, p. 278].

3. **Failing to define a copy constructor or operator= for a class with dynamically allocated memory.**

   If a C(const& C) constructor (a *copy constructor*) or operator= member function is not declared, C++ provides a default implementation for them based on memberwise initialization/assignment. For pointer members, this is bitwise copy. When objects contain pointers to dynamically allocated memory, the result of a call to a default copy constructor or operator= is multiple objects pointing to the same data. Usually this is not what is intended. (When such data sharing *is* intended, it is usually accompanied by reference counting, which itself requires a user-defined copy constructor and operator= for proper behavior.)

4. **Failing to have operator= return a const reference to the class.**

   Assignments to primitive types such as int in both C and C++ return references to their left hand arguments, as do default versions of operator= generated by C++ . This convention allows for statements of the form `w = x = y = z` to work as expected for class objects. Unfortunately, many programmers are unaware of this convention; many define operator= to return void. While some may argue that the return type of operator= is a matter of style, we believe that its return type should be based on consistency with the remainder of the language. In our view, defining operator= to return other than a reference to its class is no more defendable than is defining operator+ to mean subtraction for a numerical class.

   Unfortunately, simply returning a reference to a member of the class allows one to assign a value to the *result* of an assignment:

   ```
   (a = b) = c;    // assign b to a, then assign c to a
   ```

   As Robert Murray has noted [Mur90], "using assignment statements as lvalues is extremely unobvious and error prone. To block people from doing this, you can have operator= return a const [reference] instead of a normal [reference]."

5. **Failing to check for assignment to self in operator=.**

   Given an object `x`, the need to cope with assignments of the form `x = x` has long been recognized[Str86, p. 179]. Yet the possibility of this type of assignment is frequently overlooked – even experts forget it, e.g., [Han90, p. 234] and [Lip89, p. 267]. Its omission can be disastrous. For classes containing dynamically allocated data, the common idiom is to delete the old data (for the object on the left hand side of the assignment), dynamically allocate new space to hold the new data, then copy the data (from the object on the right hand side of the assignment) over to the new space. When the object on the left hand side *is* the object on the right hand side, however, the result of this sequence of operations is undefined.

   Regrettably, the problem of detecting that a check for `x = x` is missing is, in the general case, undecidable. However, functions that include this check almost always start out as follows:

   ```
   const C& operator=(const C& rhs) {
     if (this == &rhs) return *this;
     ...
   ```

   As a first approximation, it would suffice for a lint++ to look for a test of this form (or the equivalent check with the arguments to == reversed).

   A complicating factor is that testing for equality between `this` and `rhs` is the *wrong way* to check for `x = x` if `x` contains a duplicated base class, i.e., inherits from the same class through more than one nonvirtual path. Such situations are rare, but a programmer familiar only with single inheritance or with multiple inheritance using only virtual bases might well be unaware that what suffices in those cases may fail in the case of nonvirtual multiple inheritance. A particularly sophisticated lint++, then, might issue a warning if the address-based check for `x = x` *was* present in an assignment operator for a class that was duplicated in any other object class in the system. Readers interested in a more complete discussion of the problems involved in detecting object identity in C++ may wish to consult Adcock's article [Adc91].

6. **Problems related to the return value of a member function.**

   (a) **Returning a non-const reference to or pointer to data within an object from a const function.** As a language, C++'s meaning of "const" is "bitwise const." We firmly believe that const member functions should strengthen this contract to be "conceptually const," as well.[1] A conceptually const member function, in addition to leaving the object on which it operates unchanged (as far as the outside world can tell), must also refrain from returning "handles" through which the caller could modify the state of the object. Failure to enforce this convention renders the notion of a const object utterly useless.
   For example, consider the following skeletal class for strings:

   ```
   class String {
     private:
       char *data;   // the characters in the string
     public:
       operator char*() const { return data; }
   };
   ```

   Here, `operator` char* is bitwise const, hence the compiler will allow it to be declared as a const member function and invoked on const String objects. Yet it is clearly inappropriate to return the const object's internal pointer as a non-const, since that pointer can be used to modify the conceptual object – in this case the characters contained in the string.

   (b) **Returning a non-const reference to a data member less accessible than the function.**
   Returning such a reference essentially grants the caller direct read/write access to the underlying member. This precludes a later decision to compute instead of store a value. See the discussion under item 9 below.

7. **Overriding an inherited non-virtual function.**
   When a function *f* is defined in a base class B as non-virtual, it is an assertion that *f*'s behavior is invariant over specialization of the class; B::*f* will *always* provide correct behavior for an object, even if that object is of a derived class D. If *f* is redefined in a derived class, the assertion implicit in B::*f*'s declaration is contradicted. When this happens, something is almost always wrong. Usually the problem is either that B::*f* should have been declared virtual, or the writer of D::*f* was unaware of the fact that *f* was being inherited.

8. **Declaring a function that could lead to ambiguous calls.**
   One of the great pitfalls of C++ is that it is legal to overload a function name such that each function definition is valid, but calls to the function name may be ambiguous. This ambiguity can remain undiscovered until long after the functions are written, since some calls may be unambiguous. For example:

[1]Actually, we sometimes wish to deliberately violate bitwise constness (e.g., to allow caching) even as we maintain conceptual constness.

```
void f(int x, int y = 0);
void f(int x);

f(1, 3);      // fine, calls first f
f(4);         // ambiguous
```

If only calls of the first form are used, there will be no problems, but when the first call of the second form is used, the program will fail to compile. In this example, the ambiguity is brought about by the use of default parameter values, but ambiguity can also arise if a class inherits more than one function *f* via multiple inheritance.

9. **Declaring a data member in the public interface.**

   Declaring data members in the public interface is poor software engineering, plain and simple. It complicates the interface – users must keep track of when to use functional notation and when not to, it precludes a later decision to compute instead of store a value, and it increases coupling between classes. The use of inline member functions to get and set non-public values costs almost nothing, simplifies the public interface, reaps the benefits of functional abstraction, and decreases coupling between classes. It is difficult to envision any situation in which public data members are appropriate.

10. **Casting object pointers down the inheritance graph.**

    Casting a pointer down the inheritance graph, i.e., casting a pointer-to-Base to a pointer-to-Derived because you "know" that the pointer "really" points to a Derived object, is not an uncommon practice in C++ programs. However, it circumvents C++'s strong typing, it's brittle in the face of code changes (what you used to "know" isn't true anymore!), and it is illegal for pointers to virtual base classes. In every case such capricious casting can be replaced with safe casts through virtual functions[D'S90], and only rarely is the performance penalty serious enough to warrant concern.

11. **Passing an object by value.**

    Passing an object by value can result in a flurry of calls to constructors: for the class of the object, for its base class(es), for its members, for the members of the base classes, etc. In addition, every constructor call upon entry to the function may be matched by a destructor call upon exit. For large objects and/or objects with many base classes or members, the performance cost of these constructor/destructor calls can be staggering. They are also frequently unanticipated, especially by inexperienced programmers who are unaware of precisely what is going on when an object is passed by value, or in cases where the programmer is unaware of the true size of the object (e.g. because of typedefs). Finally, they are rarely necessary; passing a constant reference to the same object almost always suffices, with significant performance gains.

    An additional problem with passing by value is that it results in what is sometimes called "the slicing problem." Many inexperienced programmers are startled to discover that if they call a function taking a passed-by-value base class parameter *p*, and they pass in a derived class object, the specialized behavior normally exhibited by the derived class object is "sliced off," and *p* will always behave like a base class object, even when virtual functions are called on it.

12. **Having a function return a reference to an object on the stack or a dereferenced object pointer initialized by new within the function.**

In our experience, references are the most difficult concept for new C++ programmers to become comfortable with. Not quite objects and not quite pointers, they are frequently misused. Within a function, returning a reference to a local object results in the caller receiving a reference to an object which was destroyed when the function exited. Returning a dereferenced pointer initialized by new within the function results in the disappearance of the pointer, which often leads to a memory leak.

13. **Listing elements in a constructor's member initialization list in an order other than that in which they will actually be initialized.**

    The initialization order for subparts of an object, i.e., its base class data members and its own data members, is well defined, although it can be fairly complicated in the presence of virtual base classes. (Somewhat less well defined is the initialization order of static subparts, but that is immaterial to this discussion.) Class constructors can provide expressions to be used during the initialization of an object via a member initialization list that is part of the constructors' definitions. Frequently, programmers fail to understand that the order of initialization expressions in a constructor's member initialization list is ignored when initializing the object's subparts, and this can lead to bugs that are extremely difficult to diagnose.

Implementation Considerations

A lint++ for the errors just described would be of little use if it were unreasonably difficult to write or if it were prohibitively expensive to run. What follows are thumbnail sketches of how each of the errors in the previous section might be detected.

1. **Failing to match constructor calls to new with destructor calls to delete.**

   Parse each of the constructors for class *C*. Collect a list *L* of all data elements of class *C* that are initialized with or assigned the result of a call to new. Parse the destructor for class *C* and verify that all elements of *L* appear as arguments to a call to delete.

   (We are well aware of the fact that it is easy to come up with scenarios in which this approach would yield "false negatives." For example, a pointer might be initialized with the result of a call to some function *f* that itself returned the result of calling new. However, the simple approach we describe here will suffice in many cases, and it is easy to implement. We take a similar stance on the other lint++-able errors we describe in this paper.)

2. **Declaring a non-virtual destructor.**

   Parse the class declaration for class *C*. Determine whether there are any virtual functions in class *C*. Determine whether the destructor was defined and was declared virtual.

3. **Failing to define a copy constructor or operator= for a class with dynamically allocated memory.**

   Parse the class declaration for class *C*. Parse the definitions for all members of class *C*. Determine whether any of the member functions assigns to a data member of *C* the result of a call to new. Determine whether the copy constructor or assignment operation were declared.

This strategy fails to consider the possibility that friend functions can make assignments to pointer members of the class. Detecting this error in the presence of friend functions requires the parsing and analysis of several source files.

A simpler (and more conservative) algorithm would be to assume that all pointer data members may be assigned the result of a call to new, and their mere declaration would be enough to require copy constructors and assignment operators.

4. **Failing to have operator= return a const reference to the class.**

   Parse the class declaration for class *C*. Determine whether an assignment operator is declared. If so, determine whether the result is a const reference to *C*.

5. **Failing to check for assignment to self in operator=.**

   Parse the definition of operator= in class *C* if one exists. Check for the presence of a match with the regular expression for the code fragment presented earlier in this paper.

6. **Problems related to the return value of a member function.**

   (a) **Returning a non-const reference to or pointer to data within an object from a const function.** Parse the definition of each member function in class *C*. For each function returning a non-const reference or pointer to non-const object, check to see if the argument to return is a pointer to, a dereferenced pointer to, or a reference to a member of *C*.

   (b) **Returning a non-const reference to a data member less accessible than the function.**

   Parse the definition of each member function in class *C*. For each function returning a reference, check to see if the argument to return is a dereferenced pointer to or a reference to a member of *C* that is less accessible than is the member function.

7. **Overriding an inherited non-virtual function.**

   Parse the declaration of class *C* and all its base classes. For each member function declared in *C*, verify that it has a name distinct from all the non-virtual functions that *C* inherits.

8. **Declaring a function that could lead to ambiguous calls.**

   Detecting this error for member functions only can be implemented by parsing the declaration of class *C* and all its base classes. For each class, keep a list of all member function names, whether they are virtual, and their signatures. For functions with default parameters, generate multiple entries in the list, one for each different signature they can match. When building the list for *C*, add entries only for functions that do not redefine inherited virtual functions, and ensure that no entry in its list matches an entry in any of the lists built for its base classes.

   This algorithm can be extended to global functions, but it requires parsing all global functions in the program, which presumably means parsing all source files in the system. The algorithm overlooks the possibility of ambiguous function calls based on alternative sequences of type conversions to make actual parameters match formal parameters. Designing an algorithm to detect ambiguity in such cases would be considerably more complicated than what we have outlined here.

9. **Declaring a data member outside the private interface.**

   Parse the declaration of class *C*. Verify that all data members have access level private.

10. **Casting object pointers down the inheritance graph.**

    Parse the definition of each function and any previously defined classes. For each explicit cast expression, determine whether the static type of the argument to the cast is a base type of the requested type.

11. **Passing an object by value.**

    Parse the declaration of each function and any previously defined classes. Determine whether any of the arguments is an object of class type being passed by value.

12. **Having a function return a reference to an object on the stack or a dereferenced object pointer initialized by new within the function.**

    Parse the definition of each function. Determine whether the function returns a reference to an object. If so, for each return value, determine whether that value was allocated from the heap or stack by the current function.

13. **Listing elements in a constructor's member initialization list in an order other than that in which they will actually be initialized.**

    Parse the declaration of each constructor and any previously defined classes. For each member initialization list, check that the order of subparts with initialization expressions in the list is consistent with the actual order of subpart initialization.

It is useful to categorize the difficulty of these implementation strategies. We rank them based on the the minimum amount of work required by the strategy to detect a particular error:

1. Parsing of the declarations present in C++ source code.
2. Semantic analysis of the declarations present in C++ source code.
3. Parsing of the definitions present in C++ source code.
4. Semantic analysis of the definitions present in C++ source code.

Table 1 summarizes our classification of the strategies required to detect the errors described in this paper. The checkmark (√) indicates the complexity of an algorithm that will detect the corresponding error. In cases where a useful algorithm exists but is not complete (i.e., may fail to detect the error, even though it exists), a question mark (?) is used.

The fact that all of the errors described in this paper can be detected, at least some level of completeness, with no more than standard semantic analysis of source definitions encourages us that development of a lint++ would not be unduly difficult, and our experience working with and teaching the language leaves no doubt in our minds that it would be worthwhile.

The practical implications of the data in Table 1 are that errors in categories 1 and 2 could be detected by running a lint++ on C++ header files (i.e., declarations only), while errors in categories 3 and 4 would require running lint++ on implementation files.

	Complexity				Compilers
	Easier			Harder	Can Easily
Error	1	2	3	4	Detect
1				√	No
2	√				Yes
3	?			√	No
4	√				Yes
5			?		No
6a				√	Yes
6b				√	Yes
7		√			Yes
8		?			No
9		√			Yes
10				√	Yes
11		√			Yes
12				√	No
13				√	No

Table 1: Complexity of detecting different errors.

A common sentiment is that errors of this sort could (and should) be detected by current compilers with only trivial changes; we have indicated the errors that fall into this category in the last column of the table. In fact, many C++ compilers already issue warnings for some of the conditions we describe in this paper. However, a C++ compiler has a fundamentally different purpose than a lint++. A compiler is required only to detect violations of the language – its true purpose is to translate source code into object code. Programmers demand that a compiler be accurate and fast, and that the code it generates be accurate, fast, and small. A lint++, on the other hand, exists to search for constructs that are likely to have unintended consequences, and it might even presuppose that a program is syntactically correct (as verified by a compiler) when it begins its work. Such programs are typically invoked much less frequently than a compiler, and programmers are less stringent in their demands on its performance. In addition, as we discuss below, a tool like lint++ could be extended to consider issues well outside the language proper, such as verifying that design constraints expressed in some language outside C++ are satisfied within the C++ source code. Functionality such as that is well beyond the scope of traditional compilers.

A Possible Extension

An enormous number of design decisions, some of great importance, some of lesser significance, go into the development of a C++ program. Some of these decisions can be codified directly in C++, such as the fact that certain class members are not to be accessed outside the class; this is indicated by the keyword private. Other decisions can only be implied through the language, such as the fact that certain functional behavior should be invariant during class derivation; this is signified by declaring a function non-virtual (see the earlier discussion about error 7). For a great number of design decisions, however, there is no way to express the desired semantics in C++.

For example, consider a virtual member function className that is designed to return the name of the class corresponding to the current dynamic type of an object. For this function to behave properly, it *must* be redefined whenever a new class is derived. Yet there is no way to express this constraint in C++ . As a result, the constraint may easily be violated; the design is rendered ineffectual.

One way to cope with this kind of problem is to develop a metalanguage for C++ that allows source code to be annotated with explicit design constraints. Given such constraints in the source code, it would be possible to write a tool to read both the annotations and the source code, and to issue warnings if any inconsistencies were found. A simple way to do this would be to extend a lint++ to look for design constraints expressed in the form of specially formatted comments in the C++ source code. This is the approach used by lint, although the metalanguage recognized by lint is much more primitive than what we envision for a lint++. A complementary approach is to customize the behavior of a lint++ through the use of configuration files; this is the tactic being pursued by xlint, described below.

Related Work

There is no certainly no shortage of spirited opinion as to what makes "good" or "bad" C++ code, but to the best of our knowledge, this paper is the first time that a set of specific constructs have been identified that are (1) likely to be errors, and (2) detectable by a program. However, other workers have offered general guidelines as to how good classes should be developed. Riel and Carter have proposed a minimal public interface (a set of public functions) that all classes should offer [RC90], and Packstone has suggested a set of guidelines for properly implementing overloaded operators [Pac90]. Lieberherr and his colleagues have proposed the *Law of Demeter*, a language-independent rule for achieving good style in object-oriented programs [LHR88, LH89]. This law, which in C++ imposes constraints on the behavior of member functions, is formal enough to be checked by a program (as noted by its developers). Our work has some interesting relationships to this law, such as the fact that a violation of our error 6a allows a calling function to violate the Law of Demeter without even knowing it.

xlint is a program that catches some of the errors we list in this paper, but it is really designed to enforce a particular set of corporate programming guidelines, and as such is less general that the kind of tool we propose [Gre91]. However, work is underway to extend xlint so that the kinds of conditions it looks for are specified in an external file. When this work is completed, xlint will be much more configurable.

Support for formal design constraints in the form of assertions or annotations was designed into Eiffel [Mey88], has been grafted onto Ada in the language Anna [LvHKBO87], and has been proposed for C++ in the form of A++ [CL90b, CL90a]. This work, however, has grown out of the theory of abstract data types [LG86], and has tended to limit itself to formally specifying the semantics of individual functions and/or collections of functions (e.g., how the member functions within a class relate to one another). This is important work, but has so far been too limited in scope to address the kinds of design considerations we outlined above.

Summary

In this paper we argue that there is sufficient experience with C++ to justify the development of a tool that examines C++ programs for the presence of likely programmer errors, and we describe a number of common mistakes that could be detected by such a tool. We show that such a tool would be both straightforward to develop and efficient to apply. We also discuss how such a tool could be extended to detect violations of design constraints expressed in some as-yet-to-be-developed C++ metalanguage.

Acknowledgments

John Shewchuk suggested that a lint++ should check for an ordering mismatch between member initialization lists and the actual order of object subpart initialization.

Scott Meyers received support for this research from the NSF under grant DCR 8605567; by DARPA under contract N00014-83-K-0146, ARPA order 6320; and by the Digital Equipment Corporation under agreement 393. Moises Lejter received support from AFOSR contract F49620-88-C-0132, ARPA order 6426.

References

[Adc91] Jim Adcock. Is This Identity? *The C++ Report*, 3(2), February 1991.

[CL90a] Marshall P. Cline and Doug Lea. The Behavior of C++ Classes. In *Proceedings of the Symposium on Object-Oriented Programming Emphasizing Practical Applications (SOOPPA)*, pages 81–91, September 1990.

[CL90b] Marshall P. Cline and Doug Lea. Using Annotated C++ . In *Proceedings of C++ at Work - '90*, pages 65–71, September 1990.

[D'S90] Desmond D'Souza. Inheritance, Virtual Base Classes, and Casts. *The C++ Report*, 2(7), July/August 1990.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison Wesley, 1990.

[Gre91] Roger Gregory. Personal Communication. January 1991.

[Han90] Tony L. Hansen. *The C++ Answer Book.* Addison Wesley, 1990.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development.* The MIT Press, 1986.

[LH89] Karl J. Lieberherr and Ian M. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, pages 38–48, September 1989.

[LHR88] K. Lieberherr, I Holland, and A. Riel. Object-Oriented Programming: An Objective Sense of Style. In Norman Meyrowitz, editor, *Proceedings of the 1988 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '88)*, pages 323–334. ACM Press, 1988. Published as *ACM SIGPLAN Notices* 23:1, November 1988.

[Lip89] Stanley B. Lippman. *C++ Primer.* Addison Wesley, 1989.

[LvHKBO87] D. Luckham, F. von Henke, B. Krieg-Bruckner, and O. Owe. *Anna, A Language for Annotating Ada Programs: Reference Manual*, volume 260 of *Lecture Notes in Computer Science.* Springer-Verlag, 1987.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Mur90] Robert Murray. The C++ Puzzle. *The C++ Report*, 2(10), November/December 1990.

[Pac90] Jim Packstone. Heuristics for Error-Free Operator Overloading. *The C++ Insider*, 1(2), October 1990.

[RC90] Arthur J. Riel and John A. Carter. Towards a Minimal Public Interface for C++ Classes. *The C++ Insider*, 1(1), October 1990.

[Str86] Bjarne Stroustrup. *The C++ Programming Language.* Addison Wesley, 1986.

The Features of the Object-oriented Abstract Type Hierarchy (OATH)

Brian M. Kennedy

Computer Systems Laboratory
Computer Science Center
Texas Instruments Incorporated
bmk@csc.ti.com

ABSTRACT

The Object-oriented Abstract Type Hierarchy (OATH) instantiates an approach to C++ class hierarchy design that exploits subtyping polymorphism, provides greater implementation independence, and supports implicit memory management of its objects. The primary design goal of OATH was to provide an abstract type hierarchy that is consistent with the concepts being modelled by utilizing a strict subtyping approach to hierarchy design. This approach increases the polymorphism and implementation independence of code that uses OATH. The second major design goal was to provide robust garbage collection of OATH objects, fully implemented within a portable C++ class library. OATH is implemented via parallel hierarchies of internal types and "accessors". Although similar to the infamous "smart pointers", OATH accessors do not suffer many of the same problems. In particular, OATH accessors never release "dumb pointers" to the environment and fully support hierarchical argument matching. OATH accessors also offer the opportunity to "leaf" implementation classes to bypass the virtual mechanism for efficiency, when generality is not needed.

The Object-oriented Abstract Type Hierarchy (OATH) instantiates an approach to C++ class hierarchy design that exploits subtyping polymorphism, provides greater implementation independence, and supports implicit memory management of its objects. Although the core OATH library provides numerous basic types and data structures, it is the features of the hierarchy design that are most valuable to the user and developer of application-specific OATH classes. These features and their implementation are the focus of this paper.

Throughout this paper, numeric types and basic container types are used as examples. These types have the advantage that they are simple and well-understood, so no prior explanation must be offered. However, the benefits provided by OATH, and by object-oriented programming in general, are only fully realized when applied to more complex (usually application-specific) problems.

1 THE FEATURES OF OATH

The primary design goal of OATH was to provide an abstract type hierarchy that is consistent with the concepts being modelled by utilizing a strict subtyping approach to hierarchy design. This approach increases the polymorphism and implementation independence of code that uses OATH.

The second major design goal was to provide robust garbage collection of OATH objects, fully implemented within a portable C++ class library. Explicit management of dynamic memory in C++ is a burdensome and error-prone task. C++, unlike CLOS and Smalltalk, does not force the overhead of garbage collection on all programs (many of which do not need it); however, C++ does provide sufficient functionality that garbage collection can be provided in a library, so that applications that need it can have it.

OATH also provides heterogeneous container classes. Heterogeneous containers are more general and flexible than homogeneous containers, and often more natural to use. Such generality, however, requires the ability to determine the type of an object after it is removed from a container.

In C++, heterogeneity is often obtained via void*, which can point to any type and can later be cast to the appropriate type. However, such casts essentially abort the C++ type system and are thus very error-prone. The programmer must enforce some policy to ensure that objects are cast to an appropriate type.

OATH provides dynamic type determination in the form of "safe casts". A "safe cast" from an OATH type to a more derived type returns the object if it is truly of the derived type, or Nil if it is not. Nil is also a useful feature on its own. Nil can be assigned to an accessor of any type, but is itself a "non-object". Nil is similar in concept to the null pointer in C++.

2 SUBTYPING

The OATH hierarchy was designed to reflect the subtyping relationships between the types that it represents. The use of C++ inheritance for subtyping was strictly separated from implementation and code reuse. This approach to hierarchy design provides greater implementation independence, for both code inside the library and code that uses the library. The hierarchy also allows greater exploitation of subtyping polymorphism.

2.1 Subtyping v. Code Reuse

C++ classes are used to define both an abstract type (the functionality of an object) and an object implementation (the internal structure of an object). Similarly, inheritance in C++ is used for both subtyping (inheriting functionality) and code reuse (inheriting implementation). Although these two features are provided in C++ with the same mechanism, they are distinctly different concepts [Amer90].

Code reuse is a powerful feature of C++; however, it is a poor basis for object-oriented design. A type hierarchy should be designed to reflect the behavior of the objects being modelled. It should not be designed to reflect the most convenient computer representation of the objects.

For example, consider `rational` and `integer`, common multi-precision numeric types. Many class libraries have been proposed and/or implemented (e.g. Smalltalk [Gold83], NIH OOPS [Gorl87], libg++ [Lea88]) that define `rational` as a sibling class of `integer` implemented as a pair of `integers`. Such a definition is simple, exploiting well the power of code reuse via composition. Unfortunately, this definition does not correspond to the mathematical concepts being modelled. Mathematically, all integers are rational numbers -- integer is a subtype of rational.

Some libraries (e.g. CLOS [Stee90]) do implement `integer` as a subtype of rational (`fraction`); however, these same libraries implement real numbers and complex numbers as siblings of rational. Mathematically, all rational numbers are real numbers, and all real numbers are complex -- rational should be a subtype of real which should be a subtype of complex.

The primary goal of OATH is to provide a meaningful abstract type hierarchy: a hierarchy of behavioral specifications that correspond to the concepts being modelled (see Figure 1). Given a consistent abstract type hierarchy, implementation classes (in italics in Figure 1) can be added at the leaves of the hierarchy to implement the behavior of the

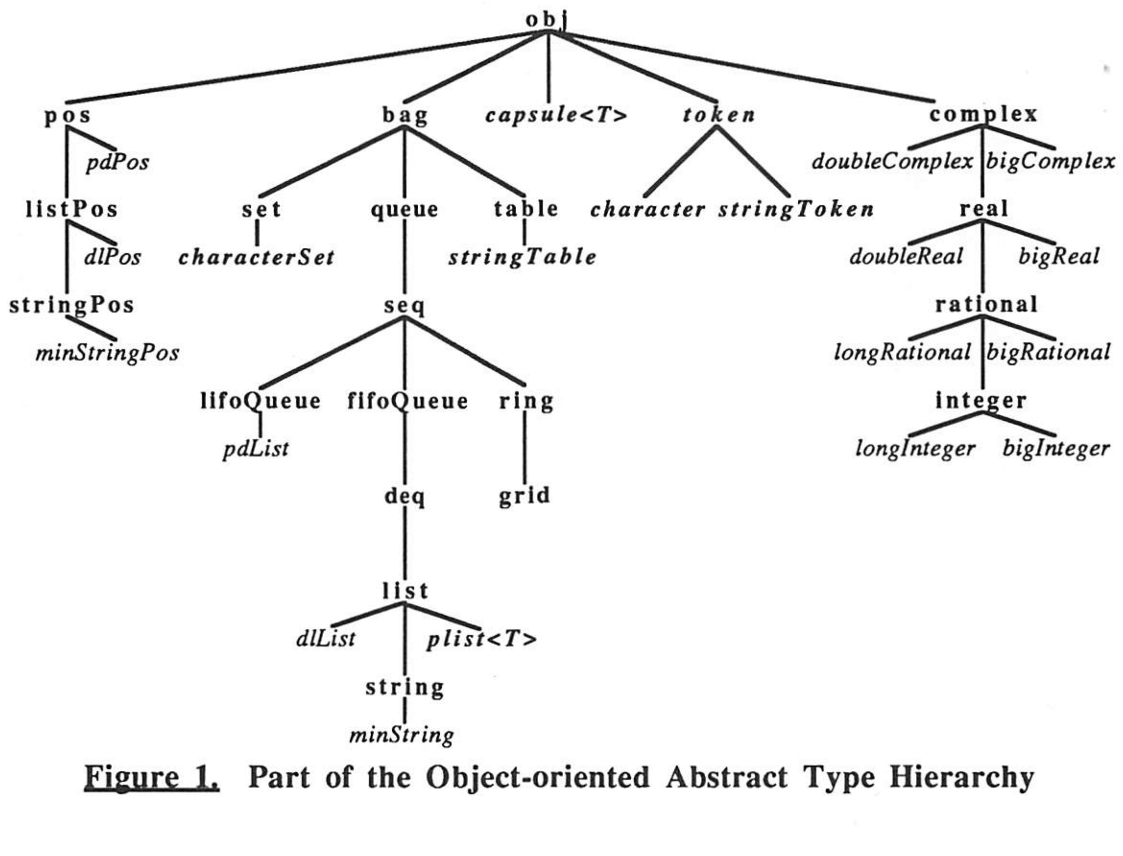

Figure 1. Part of the Object-oriented Abstract Type Hierarchy

abstract types (in bold). Code reuse can be exploited at this phase, but should not enter into the design of the abstract type hierarchy.

For instance, in OATH `integer` is derived from `rational`, which is derived from `real`, which is derived from `complex`. These are all abstract types. The type `integer` can be implemented in more than one way. The implementation type `bigInteger` is multiple-precision, whereas `longInteger` is implemented as a `long`. Similarly, `rational` can have several implementations: `bigRational` is a pair of `bigIntegers` and `longRational` is a pair of `longs`. Note that code reuse via composition was exploited without affecting the abstract type hierarchy.

In addition to the conceptual consistency of the hierarchy, this subtyping approach to hierarchy design clearly separates the implementations from the behavioral specifications. This simplifies code maintenance, allows alternate implementations to be added later, and provides a level of implementation independence in code that uses the library. For example, a user of `integer` need not know the implementation. It may be implemented as a `long`, as two `longs`, as an array of `longs`, or as a linked-list of `longs`. Furthermore, another implementation of integer may be added later without rewriting the hierarchy or the code that uses the hierarchy.

2.2 Subtyping Polymorphism

This hierarchy design also provides a great deal of subtyping polymorphism. For example, any code written using `complex` numbers will work with `reals`, `rationals`, and `integers`. Similarly, any code written using `bags` will also work with `queues`, `lists`, and `strings`.

For example, consider the common tree search algorithm which records in a structure all possible paths (decisions), and then chooses one. Later it may return to the structure in order to try an alternate path. Separate algorithms could be coded for breadth-first, depth-first, and prioritized searches.

In contrast, a single search algorithm could be coded which is polymorphic on the alternate path structure. A `queue` has sufficient functionality (insertion and extraction) to implement the search algorithm, without knowing exactly what queueing paradigm is used. If this algorithm is passed a `lifoQueue` (last-in first-out queue, or stack), then it will perform a depth-first search. If it is passed a `fifoQueue`, (first-in first-out queue), then it will perform a breadth-first search. If it is passed a prioritized queue, then it will perform a prioritized search.

2.3 Subtyping Domain

For `bags` (OATH container classes), there are two forms of subtyping: increasing functionality and restricting domain. For instance, the OATH `string` is a subtype of `list` that can contain only `characters`. It also has additional functionality, such as <, <=, >, >=, uppercase, lowercase, and hash, which make sense because of `string`'s restricted domain.

Bag and its descendants were designed to allow this restriction of domain, or filtering. Any object may be inserted into a bag; however, depending upon the subtype of that bag, the object may fall through (conceptually as if the bag were a sieve or filter).

This ability to subtype domain can be used to increase polymorphism. For instance, returning to the search algorithm above, a filtering lifoQueue could be passed to the algorithm in order to completely ignore paths that were undesirable. In this way, a completely different search can be made with the same search algorithm.

Although OATH bags were designed to support applications that need heterogeneous containers, there are many applications that do not need heterogeneity. Homogeneous containers provide simplicity of expression, increased type safety, and execution efficiency. Homogeneous class implementations can be conveniently provided as subtypes of their heterogeneous counterparts via parameterized types. For instance, plist<complex> may be an implementation of list that holds only complex numbers. (Note: this is conceptual -- parameterized types have not yet been used with OATH).

2.4 Subtyping and Execution

This subtyping approach to hierarchy design can effect execution efficiency. Since most functionality is defined as virtual functions, separate from the implementations, out-of-line virtual calls are common when utilizing the generality of the abstract classes. However, the design of the OATH accessors (described below) allows the definition of "leaf" implementations that allow more efficient execution by bypassing the virtual mechanism.

3 OATH ACCESSORS

Users of OATH do not access the objects directly; OATH objects can only be accessed through OATH "accessors". For each OATH type there is a corresponding accessor type. The accessors can be initialized and assigned OATH objects to access (analogous to C++ pointers). However, any other operation on an accessor is applied directly to the abstract object that it accesses (analogous to C++ references). Thus, accessors can be used as if they were the objects themselves, but assigned and passed as function parameters as if they were pointers.

Since the OATH accessor lies between pointers and references, a new but similar syntax would be nice. For instance, "@" could be used instead of "*" or "&":

```
list@ L;
```

However, OATH is intended to be a C++ library, not a new language. So, the OATH accessor types are suffixed with a capital letter "A":

```
listA L;
```

Analogous to pointers, constructing a listA constructs only an accessor, not the list itself. In the declaration above, L is initialized to Nil. Given an existing listA K, then L could be initialized to access the same list object that K accesses:

```
listA L = K;
```
Note that both K and L access the same list. To make a new OATH object, a "make" function must be called. For instance, to initialize L to access a copy of the list accessed by K, the function makeCopy can be used:
```
listA L = K.makeCopy();
```
To make an object from scratch, an implementation type must be chosen and its "make" function, a static member of the accessor class, must be called. For instance,
```
listA L = dlListA::make();
```
makes an empty dlList (doubly-linked list) and assigns it to the list accessor L.

In addition to static "make" functions, the accessor types also have a static member function is, which is the OATH "safe cast". For instance, given a bagA B,
```
listA L = listA::is(B);
```
attempts to "safe cast" B to a list and assign it to L. If B is not really a list, then is(B) will return Nil.

3.1 Parallel Hierarchies

OATH is implemented as two parallel hierarchies: the accessor type hierarchy and the internal type hierarchy. The internal types contain the object representation (the data members) and the virtual functions. The accessor types contain all of the externally accessible functions of the abstract types. These functions often do little more than call the appropriate virtual function(s) in the internals hierarchy. The accessors have a single data member, a pointer to an object in the internals hierarchy. Thus, accessors are one-word structures which can be held and passed in registers, and otherwise optimized like the built-in pointer types.

Efficient use of the accessors is natural. For instance, there is no significant cost in passing or returning accessors by value. Further, construction and destruction is equivalent in cost to re-assignment, so placing an accessor in a loop does not have hidden overhead.
```
stringPosA P = S.makePos();
for(; P(); ++P)
   {characterA C = *P;
    // do something with C
   }
```
Unlike many multi-word C++ classes, there is no extra cost in placing the characterA declaration within the for-loop (where it belongs). Thus, code can be written fairly naturally with accessors without incurring unforeseen inefficiencies.

3.2 Accessors v. Smart Pointers

OATH accessors are similar in concept to smart pointers, which have been proposed [Stro87] and implemented [Wang89][Edel90] many times before. However, OATH accessors offer some significant advantages over smart pointers.

First and foremost, OATH accessors never release "dumb" pointers outside of their member functions. This will have important consequences on garbage collection when compiled with C++ compilers that destruct temporary objects as soon as possible (as permitted by the current draft standard and [Elli90]).

Smart pointers overload operator -> to return a dumb pointer to its "internal" object. Such a definition is convenient, since it makes all members of the internal object immediately available through the smart pointer. However, this definition is problematic. Consider,

```
O2 = O1->makeCopy()->transform();
```

The desire is to set `O2` to a transformed copy of `O1`. `O1->` returns a dumb pointer which is used as this for the member function `makeCopy()`. The member function `makeCopy()` returns a smart pointer to a new object that is a copy of `O1`. The compiler will create a temporary object to hold that smart pointer. The operator `->` applied to the temporary yields a dumb pointer to the new copy. At this point, prior to the call to the call of `transform()`, the compiler can destruct the temporary that holds the smart pointer. It can do this because it no longer needs the smart pointer once it has obtained the dumb pointer returned from operator `->`. Since the temporary was the only smart pointer referencing the new copy, that copy may be collected (destruction of the smart pointer may cause its immediate destruction, or the invocation of `transform()` could cause a garbage collection). (Note that many current C++ compilers keep temporary objects alive past the end of the expression, so the above has not been a problem. Future compiler implementations will probably not.)

In contrast, OATH accessors define the type interface. So, a member function invocation on a temporary OATH accessor invokes a member function of that object, thereby guaranteeing that the temporary will exist until the end of the function. All uses of "dumb" pointers to internal OATH objects are dynamically within a call to an OATH accessor member function.

OATH accessors also offer the advantage of reference semantics, which makes their use much more natural in the presence of overloaded operators. Finally, since OATH accessors are defined in a parallel hierarchy, they can be assigned and passed as arguments to overloaded functions naturally, obeying the implicit conversion preferences of the inheritance hierarchy. Definition of user-defined conversion operators are often necessary with smart pointers: preference based upon depth in the hierarchy is not considered when user-defined conversions are invoked.

3.3 Accessors and Execution

The parallel hierarchy structure of OATH allows an implementation type to be defined as a "leaf", such that when it is used directly it bypasses the virtual mechanism. Whether or not to define an implementation class as a leaf is a direct trade-off between execution efficiency and reusability via derivation.

For instance, it may be desirable to have an efficient multiple-precision integer class, bigInteger. The following expression with general OATH integers,

```
I1 += I2 * I3;
```

would require two virtual function invocations. To prevent this for the leaf class bigInteger, the accessor class `bigIntegerA` redefines operators += and * to call the internal function directly, bypassing the virtual mechanism, by using a scoped function call. If both the internal functions and the accessor functions are inline, then the whole expression can be coded inline when `bigIntegerAs` are used:

```
          bigIntegerA BI1, BI2, BI3;
          // code here
          BI1 += BI2 * BI3; // no virtuals
```

Thus, bigInteger can be used as an implementation of integer, rational, real, and complex via the general virtual mechanism in code that needs generality. Alternately, for code that needs efficiency, bigInteger can be used specifically and the virtual functions can be bypassed. The disadvantage of defining an implementation class as a leaf is that it cannot be easily reused via derivation.

4 LIBRARY-BASED GC

Explicit management of objects allocated in freestore is notoriously error-prone and generally completely disjoint from the algorithm that is being coded. One of the major features of OATH is a library-based garbage collection mechanism for OATH objects. This mechanism is a hybrid reference counting and marking algorithm capable of collecting all garbage (including circular references).

4.1 Reference Counting

The internal representations of OATH objects are always allocated in freestore and always accessed via OATH accessors. Thus, it is simple to maintain accurate reference counts [Knut73] on OATH objects. When an accessor is assigned an OATH object, it increments the reference count of the object being assigned, and it decrements the reference count of the object that it previously accessed. Construction and destruction increment and decrement the reference count, respectively.

4.2 Modes

The programmer can select one of four garbage collection modes at compile-time: no GC, incremental GC, stop-and-collect, or combined. No GC mode eliminates the overhead associated with garbage collection. This mode is suitable for short-lived programs and programs that make few objects during execution.

Incremental GC mode maintains reference counts on the objects. If a reference count is zero after being decremented, then the object is deleted. This mode is more convenient than stop-and-collect, since the programmer need not decide when to invoke garbage collection. However, circularly-referenced garbage will not be collected. This mode is probably best suited for programs that do not produce circular references or do not live long enough that the lost storage will matter.

Stop-and-collect mode will maintain reference counts, but will only collect when the programmer calls the function `objA::collectGarbage(int)`. The int parameter specifies "quick" collection or "full" collection. Full collection will collect circularly-referenced garbage, but can be significantly more time consuming. An extra word of storage per object is used in stop-and-collect mode. This mode is typically preferred for programs that produce circular references, but do not overflow from physical memory into virtual memory.

The combined GC mode collects incrementally and `collectGarbage()` can be invoked to collect circularly-referenced garbage. However, this mode requires two extra words of storage per object. This is the best mode for programs that are long-lived or utilize virtual memory.

In stop-and-collect and combined modes, all OATH objects are linked together so that they can be traversed by `collectGarbage()`. The extra storage required by these two modes is due to the links. In stop-and-collect mode, the objects are singly-linked (hence, one extra word per object). To facilitate incremental collection in combined mode, the objects are doubly-linked (hence, two extra words per object).

4.3 Collecting Circular References

Reference counting is an efficient and robust way to implement library-based garbage collection. However, circularly-referenced garbage cannot be collected from reference counting alone.

Traditional two-pass marking garbage collectors maintain a set of root pointers. Any object that is unreachable from the set of root pointers is garbage. To identify all garbage, the first pass starts from each root pointer and marks all objects that can be reached. The second pass over the objects simply collects all unmarked objects.

OATH accessors can be split into two groups, "internal accessors" and "root accessors". Internal accessors are accessors that are held by OATH objects -- these are the accessors that cause circular references. Root accessors are accessor objects held by the application, on the stack, in static storage, or as members of non-OATH objects. For the two-pass marking algorithm above, the set of root pointers would be the root accessors. Maintaining a record of the root accessors, excluding internal accessors, would be difficult, making use of accessors very expensive. In contrast, OATH uses a three-pass algorithm which requires only the reference counts ([Chri84] proposes a similar five-step algorithm).

At the beginning of the first pass, the reference counts include references due to both root and internal accessors. During the first pass, the virtual function `clearReferences()` is called on each object. This function clears the mark (a one-bit flag in each object) and then calls `deref()`, which decrements the reference count, on each object that it references. At the end of this first pass, all reference counts due to internal accessors have been removed. The reference counts that remain are due to root accessors.

The second pass calls the virtual function `setReferences()` on each object that has a non-zero reference count (is referenced by a root accessor) and is not marked (has already been visited). The function sets the mark flag and then calls `ref()`, which increments the reference count, and, if unmarked, `setReferences()` on each object that it references. At the end of the second pass, all reference counts due to internal accessors that are reachable from root accessors have been restored. The reference counts of circularly-referenced garbage will remain zero. A final pass is then made to delete each object with reference count equal to zero.

This three-pass marking algorithm is clearly more expensive than the traditional two-pass algorithm; however, it is the same order of complexity. Furthermore, this cost is only incurred by programs that need to collect circularly-referenced garbage. Incremental collection, which is quite efficient, will be sufficient for most programs.

ACKNOWLEDGEMENT

OATH is a result of many long, often heated, discussions with Larry Spry, whose insights and experience proved quite valuable.

REFERENCES

[Amer90] P. America and F. van der Linden, "A parallel object-oriented language with inheritance and subtyping", *ECOOP/OOPSLA '90 Proceedings*, 21-25 Oct 1990, p. 161-168.

[Chri84] T. W. Christopher, "Reference Count Garbage Collection", *Software -- Practice and Experience*, 14(6), p. 503-507, June 1984.

[Edel90] D. R. Edelson, *Dynamic Storage Reclamation in C++*, Master's Thesis, University of California at Santa Cruz, UCSC-CRL-90-19, June 1990.

[Elli90] M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, 1990.

[Gold83] A. Goldberg and D. Robson, *Smalltalk-80 The Language and its Implementation*, Addison-Wesley, 1983.

[Gorl87] K. E. Gorlen, "An object-oriented class library for C++ programs", *Software -- Practice and Experience*, v17(12), Dec 1987, p. 899-922.

[Knut73] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Addison-Wesley, 1973.

[Lea88] D. Lea, "libg++, The GNU C++ library", 1988 USENIX C++ Conference, p. 243-256.

[Stee90] G. L. Steele Jr., *Common Lisp The Language*, Digital Press, 1990.

[Stro87] B. Stroustrup, "Possible Directions for C++", 1987 USENIX C++ Workshop, p. 399-416.

[Wang89] T. Wang, *The "MM" Garbage Collector for C++*, Master's Thesis, California Polytechnic State University, 1989.

The Separation of Interface and Implementation in C++

Bruce Martin

Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, California 94304
martin@hplabs.hp.com

abstract

A C++ *class* declaration combines the external interface of an object with the implementation of that interface. It is desirable to be able to write *client* code that depends only on the external interface of a C++ object and not on its implementation. Although C++ encapsulation can hide the implementation details of a class from client code, the client must refer to the class name and thus depends on the implied implementation as well as its interface.

In this paper, we review why separating an object's interface from its implementation is desirable and present a C++ programming style supporting a separate interface lattice and multiple implementation lattices. We describe minor language extensions that make the distinction between the interface lattice and implementation lattice apparent to the C++ programmer. Implementations are combined using standard C++ multiple inheritance. The operations of an interface are given by the union of operations of its contained interfaces. Variables and parameters are typed by interfaces. We describe how a separate interface lattice and multiple implementation lattices are realized in standard C++ code.

1.0 Introduction

The *class* construct of the C++ language[10] mixes the notion of an interface to an object with the implementation of it. This paper demonstrates how to separate interface from implementation in C++.

An *interface* declares a set of operations. An object typed by that interface guarantees to support those operations. An *implementation*, on the other hand, defines *how* the object supports those operations, that is an implementation defines the representation of the object and a set of algorithms implementing the operations declared in the interface.

An interface may contain other interfaces and add new operations. An object whose type is given by the expanded interface supports the *union* of the operations of the contained interfaces and the new operations. An object supporting the expanded interface may be accessed by code expecting an object supporting one of the contained interfaces. If an interface is defined by combining multiple interfaces, an interface lattice results.

Similarly, an implementation may be provided in terms of other existing implementations.

That is, rather than providing implementations for all of the operations of an interface, an implementation may *inherit* some code and representation from other implementations. In the presence of multiple inheritance of implementations, an implementation lattice results.

If interface is separated from implementation, the interface lattice of an object need not be the same as the implementation lattice of the object. That is, the structure of the interface lattice need not be equivalent to the implementation lattice. Furthermore, there may be several different implementation lattices supporting the same interface.

1.1 Why separate interface from implementation?

Separating interface from implementation is desirable for achieving flexible, extensible, portable and modular software. If client code[1] depends only on the interface to an object and not on the object's implementation, a different implementation can be substituted and the client code continues to work, without change or recompilation. Furthermore, the client code continues to work on objects supporting an expanded interface.

Snyder describes in [9] how combining interface and implementation in a single class construct violates encapsulation. Snyder demonstrates that changing an object's implementation affects clients of that object when inheritance is used both for reusing code and for subtyping.

Our primary motivation for separating interface and implementation in C++ is to cleanly map C++ on to a system of distributed objects. In a distributed environment, allowing multiple implementation lattices for an interface is essential. Interfaces are *global* to the distributed environment, while implementations are *local*. For a distributed program that crosses process, machine and administrative boundaries, maintaining a single implementation of an interface is difficult, if not impossible. However, as described in [6], it is feasible in an RPC based system to maintain a global space of interfaces.

1. We refer to code invoking an operation on some object as *client code*. The term *client* does not necessarily denote distributed computing.

1.2 C++ classes

A C++ *class* combines the interface of an object with the implementation of that interface. Although C++ encapsulation can hide the implementation details of a class from client code, the client code must refer to the class name and thus depends on the implied implementation.

Consider the C++ class, `stack`, in figure 1. The stack abstraction is implemented by an array, `elements`, and an integer, `top_of_stack`. Both the abstraction and the implementation are named, `stack`. Client code that declares variables and parameters of class `stack` identifies both the abstraction and the implementation.

```
class stack {
     int elements[MAX];
     int top_of_stack;
  public:
     void push(int);
     int pop();
};
```

Figure 1. A C++ class, `stack`

A C++ *derived class* may add both to the interface of an object and to its implementation. That is, the derived class may add new public member functions and private members. The single class construct implies a single combined interface and implementation lattice.

Consider the derived class, `counted_stack` of figure 2. It expands the public interface by adding a member function, `size()` and it inherits the private members, `elements` and `top_of_stack`. It is impossible to be a counted_stack without also containing the array, elements.

```
class counted_stack:
   public stack {
      int no_items;
   public:
      int size();
      void push(int);
      int pop();
};
```

Figure 2. A derived class, counted_stack

C++ 2.0 has added *pure virtual functions, multiple inheritance* and *virtual base classes* to the language. This paper shows how those constructs can be used to support a separate interface lattice and multiple implementation lattices.

1.3 Related work

Languages such as Ada and Modula 2 explicitly separate the interface to a program module from the implementation of it. These languages do not have mechanisms for inheriting implementations.

The Abel project[1] at Hewlett-Packard Laboratories has explored the role of interfaces in statically typed object-oriented programming languages. Abel interfaces are more flexible than those described here for C++. In particular, the return type of an operation may be specialized and parameters generalized in an extended interface. C++ requires them to be the same.

Several object-based distributed systems ([7], [2], [3], [8]) use or extend C++ as the programmer's interface to the distributed objects. In such systems, interfaces are not explicit but rather considered to be the public member function declarations of a C++ class. As such, interface and implementation lattices must have the exact structure at all nodes in the distributed system. This paper demonstrates how such C++ distributed systems can have an explicit, global interface lattice and multiple, local implementation lattices.

2.0 Separation Model

This paper presents a model for C++ programs in which an interface lattice can be supported by different multiple implementation lattices. We describe the model in terms of some minor language extensions that have been implemented in a preprocessor producing standard C++ 2.0 code. The language extensions make the separation between interface lattice and implementation lattice apparent to the C++ programmer; they also make our description clearer. However, the model could be viewed as a C++ programming style and programmers could write the C++ we describe in section 3.0 directly. The language extensions enforce the style.

Throughout the paper, we use an example of a *bus stop*. The interface *BusStop* is given by combining the interface *PeopleQueue* with the interface *Port.*

2.1 Interfaces

Figure 3 gives the interface to a queue of people. The interface declares three operations: enq adds a person to the queue, deq removes and returns a person from the queue and size returns the number of people in the queue.

```
interface PeopleQueue {
   enq(person *);
   person *deq();
   int size();
};
```

Figure 3. The interface to a queue of people

The interface represents a contract between client code invoking an operation on an object meeting the PeopleQueue interface and code implementing the interface. The contract states that the object will support the three operations specified in the contract. It says nothing, however, about the implementation of those three operations.

Similarly, figure 4 gives the interface to a `Port`. A `Port` represents a place where a vehicle departs at some time for some destination. The `dest` operation returns a reference to an object meeting the `city` interface and the `departs` operation returns a reference to an object meeting the `time` interface.

```
interface Port{
   city *dest();
   time *departs();
};
```

Figure 4. The interface to a port

An operation declaration gives a name and the types of parameters and return values. Parameters and return values are either C++ primitive and aggregation types or they are references to objects supporting an interface. They are *not* typed by a C++ implementation class. For example, in the `PeopleQueue` interface of figure 3, the `enq` operation takes as a parameter a reference to an object supporting the `person` interface, the `deq` operation returns an object supporting the `person` interface and the `size` operation returns an integer value.

An interface declares the *public* interface to an object; there are no `public`, `protected` or `private` labels, as there are in C++ classes.

Interfaces can be combined and expanded. Figure 5 defines a `BusStop` interface in terms of a `PeopleQueue` and a `Port`. The declaration states that a `BusStop` is a `PeopleQueue` and it is a `Port`. An object meeting the `BusStop` interface supports all of the operations defined by the `PeopleQueue` and by the `Port`; it can be used in any context expecting either an object meeting the `PeopleQueue` interface or one meeting the `Port` interface. In addition, a `BusStop` supports the `covered` operation. (The `covered` operation is true if the bus stop is covered.)

```
interface BusStop :
   PeopleQueue, Port {
      boolean covered();
};
```

Figure 5. The interface to a bus stop

Figure 6 presents a graphical representation of the interface lattice for a `BusStop`.

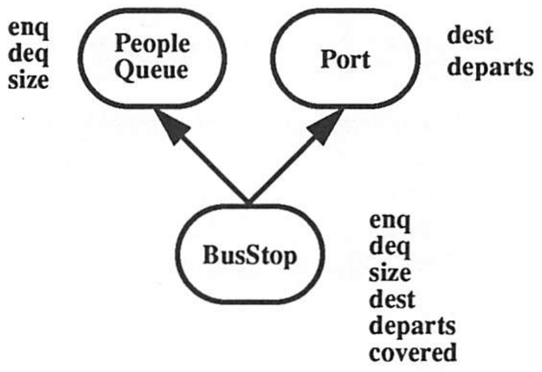

Figure 6. Interface lattice for a bus stop

The containment of interfaces is not as flexible as discussed in [1]. Operations are simply the declarations of member functions in the C++ sense. Refinement of parameter or result types is not supported. As in C++, an operation whose name matches another operation but whose parameters differ is considered to be overloaded.

Ambiguities cannot result when combining multiple interfaces; the operations of an interface are declarations, not definitions.

2.2 Implementations

Interfaces alone do not result in executing programs. Programmers must also provide *implementations* of an interface. Figure 7 gives an implementation, named `linked_people`, of `PeopleQueue`. It uses a linked list to repre-

```
class linked_people
   implements PeopleQueue {
      recptr *head, *tail;
   public:
      linked_people() {
         head=tail=NULL;
      }
      enq(person *p);
      person *deq();
      int size();
};
```

Figure 7. A linked list implementation of a queue of people

sent the queue. Figure 8 gives an implementation, named `people_buffer`, of `PeopleQueue`. It uses an array to represent the queue.

Notice in figures 7 and 8 that a class is declared to be an implementation of an interface using the `implements` keyword. Multiple implementations of the same interface can exist in a single C++ program; `people_buffer` and `linked_people` both implement the `PeopleQueue` interface.

Classes provide algorithms implementing the operations declared in the interface and may declare state and local member functions. Local member functions are called in implementation code. Implementations may also define constructors and destructors for the implementation. A constructor is inherently implementation dependent. For example, the constructor for the `people_buffer` in figure 8 takes an integer argument indicating the upper bound on the size of the buffer, while the constructor for the `linked_people` takes no arguments.

```
class people_buffer
   implements PeopleQueue {
      person **buf;
      int last;
   public:
      people_buffer(int sz) {
         last=0;
         buf = new person *[sz];
      }
      enq(person *p);
      person *deq();
      int size() {return last;}
};
```

Figure 8. An array implementation of a queue of people.

Notice that the parameter types of operations are given as other interfaces, not implementations. Thus, it would be impossible to define an operation that took a `linked_people` as a parameter. Similarly, local and member variables are typed by interface, not implementation. By doing so, client code is independent of a particular implementation.

Implementations may *reuse* other implementations. For example, figure 9 gives an implementation of `BusStop`, named `muni_stop`, suitable to represent municipal bus stops. It reuses the implementation of `linked_people`.

```
class muni_stop
   implements BusStop
   reuses public: linked_people {
      boolean shelter;
   public:
      muni_stop(boolean cov){
         shelter=cov;
      }
      city *dest();
      time *departs();
      boolean covered() {
         return shelter;
      }
};
```

Figure 9. Municipal bus stop implementation of the bus stop

Figure 10 gives a graphical representation of the `muni_stop` implementation.[2]

Figure 11 gives a different implementation of `BusStop`, named `inter_city`. It reuses

2. We use ovals to represent interfaces and rectangles to represent implementations in all graphics.

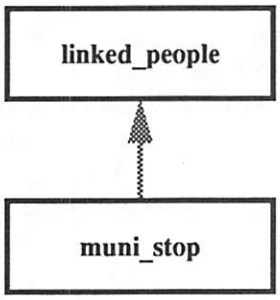

Figure 10. Graphical representation of the muni_stop **implementation**

both the people_buffer and the pair[3] implementations.

Figure 12 gives a graphical representation of the inter_city implementation.

Notice that there are multiple implementation lattices in figures 10 and 12, that they have different structures and that the *implementation* lattice of figure 10 is not structurally the same as the *interface* lattice of figure 12. Without the separation of interface and implementation this would not be possible.

Implementations reuse other implementations according to C++ inheritance semantics. A class declares the operations it will implement and inherits the ones it does not. Thus, for example, the muni_stop implementation of the BusStop interface given in figure 9 declares and implements the dest, departs and covered operations but reuses the enq, deq and size functions from the linked_people class. On the other hand, the inter_city implementation declares and implements only the covered operation and reuses the others from people_buffer and pair.

```
class inter_city
   implements BusStop
   reuses public: pair,
      public: people_buffer {
   public:
      inter_city();
      boolean covered() {
         return true;
      }
};
```

Figure 11. Intercity implementation of the bus stop

3. The pair implementation of the Port interface is left to the reader as an exercise.

Ambiguities are resolved in the C++ way, by the programmer. If a class inherits ambiguous member functions from multiple classes, the class must also declare the member function and provide an implementation of it. There is, of course, no ambiguity caused by interfaces -- the interface lattice is separate.

Programmers control the visibility of the declarations in the implementation lattice using public, private and protected. However, member functions that implement operations declared in the interface lattice are, by definition, public.

2.3 Object instantiation

Implementations are named when an object is instantiated by the new operator or allocated on the run time stack. However, to promote modularity and flexibility, this code should be isolated. Client code that refers to an object via variables typed by the object's implementation, rather than by one of its supported interfaces, creates unnecessary dependencies on imple-

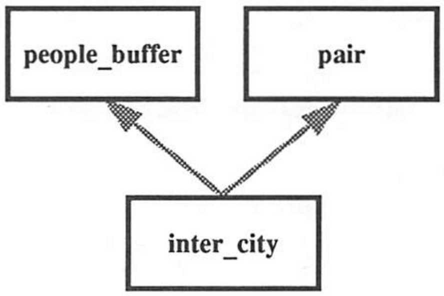

Figure 12. Graphical representation of the inter_city **implementation**

mentation; the client code can only be used with objects of that implementation.

Most code should refer to an object using variables typed by interface. For example, if an instance of the `muni_stop` implementation is created like this:

```
BusStop *bs = new muni_stop(false)
```

all further references to it will be typed by its interface, `BusStop`, not by its implementation.

Similarly, when an object is allocated on the run time stack, it should be referred to by an interface it supports. A `muni_stop` implementation and a `inter_city` implementation are created on the run time stack in figure 13. The identification of the implementation should be isolated to these declarations. The `simulate` function is defined to operate on two objects supporting the `BusStop` interface, independent of implementation. The `simulate` function is a client of the objects. It cannot see anything about the implementations; the types of the parameters do not name implementations.

```
    muni_stop ms(false);
    inter_city ic;
    simulate(BusStop*,BusStop *);
          :
    simulate(&ms, &ic);
          :
```

Figure 13. Naming implementations when allocated on run time stack

2.4 Design Goals

The design of the separation model was constrained by three practical requirements.

First, the separation model needed to be easily realized in C++. This meant that at some level, the separation model had to be a programming style. This constrained the model. For example, the model allows an object of an extended interface to be accessed by code expecting an object of a contained interface but as shown in [1], the notion of a contained interface is stronger than it need be.

Next, the design of the separation model was also constrained by a desire to preserve the C++ inheritance model for reusing implementations. The rules for when to declare a member function in a class, the C++ approach towards ambiguities and the definitions of virtual and non-virtual base classes were preserved for implementations.

Finally, what the separation model adds should be simple. To use the separation model, the C++ programmer must learn

- that an extended interface supports the union of the operations of its contained interfaces,
- that variables and parameters should be typed by interfaces not by implementations,
- that code instantiating an object names an implementation and should be isolated
- and that all class member functions implementing operations declared in interfaces are public.

3.0 Realization in C++

We now discuss how to realize the separation model in C++ and some of the problems we encountered in realizing it. Although we describe this in terms of the behavior of our preprocessor for the language extensions given in section 2.0, it could be viewed as a programming style and the C++ programmer could write this code directly.

The presentation assumes some understanding of C++ implementation strategy, particularly for multiple inheritance. It is too detailed to repeat here; we refer the reader to [11].

3.1 Interfaces

An interface is translated to a C++ class of the same name containing only pure virtual functions. Pure virtual functions, those defined with the odd `=0`, indicate the C++ class will *not* provide an implementation of the function. A C++ class with a pure virtual function cannot be instantiated via the `new` operator or on the stack. These are the desired semantics for interfaces. Figure 14 gives the `Port` interface of figure 4 translated to standard C++.

```
class Port {
   public:
      virtual city *dest()=0;
      virtual time *departs()=0;
};
```

Figure 14. C++ representation of the `Port` interface

Interfaces contain *only* pure virtual functions; C++ allows pure virtual functions to be declared in a class that also contains implementation. Doing so breaks the strict separation of interface and implementation.

3.2 Combining Interfaces

The separation model defined the operations of an extended interface to be the union of the operations in its contained interfaces and the added operations. Furthermore, the separation model allows an object supporting an interface to be accessed in a context expecting an object of one of the contained interfaces.

Combining interfaces by inheriting the C++ abstract classes representing the interfaces almost works. The *isa* relationship is provided by inheriting C++ abstract classes. However, C++ 2.0 does not allow pure virtual functions to be inherited and requires them to be redeclared in derived classes. When translating an interface declaration, the preprocessor generates the union of the pure virtual functions in the derived abstract classes. Figure 15 gives the `BusStop` interface of figure 5 translated to standard C++. Notice that the BusStop is declared to be a derived class of both `PeopleQueue` and `Port` but redeclares the pure virtual functions from both.

```
class BusStop :
      public virtual PeopleQueue,
      public virtual Port {
   public:
      virtual boolean covered()=0;
      virtual int enq(person *)=0;
      virtual person *deq()=0;
      virtual int size()=0;
      virtual city *dest()=0;
      virtual time *departs()=0;
};
```

Figure 15. C++ representation of the `BusStop` interface

The C++ abstract classes representing interfaces are inherited as virtual base classes.

As discussed in [11], using non-virtual base classes may make sense for some implementations. It does not for interfaces. Virtual base classes are semantically closer to the separation model's notion of interface combination.

Using virtual base classes to represent interfaces results in an "inconvenience" for the programmer. Without virtual base classes, type casting is implemented as pointer arithmetic. Programmers are allowed to do unsafe type casting from a reference to a base class to a reference to a derived class because it requires a simple address calculation. However, C++ represents virtual base classes as a pointer in the derived class to the virtual base class. Casting from a derived class to a virtual base class is not a simple address calculation but instead follows the pointer to the virtual base class. Casting from a virtual base class to a derived class is not supported.

Some might consider the restriction on type-unsafe casting to be a benefit of the representing interfaces as virtual base classes! However, C++ programs often do need to cast from a base class to a derived class. Achieving this in a type safe fashion requires associating type information at run time with objects such as proposed in [4]. Basically, implementations can return the addresses of the "interface part" being requested. One possible result, of course, is that the implementation does not support the requested interface and the cast fails.

3.3 Implementations

An implementation is represented as a C++ derived class of the interface it implements. An implementation is also a C++ derived class of the C++ classes representing the reused implementations.

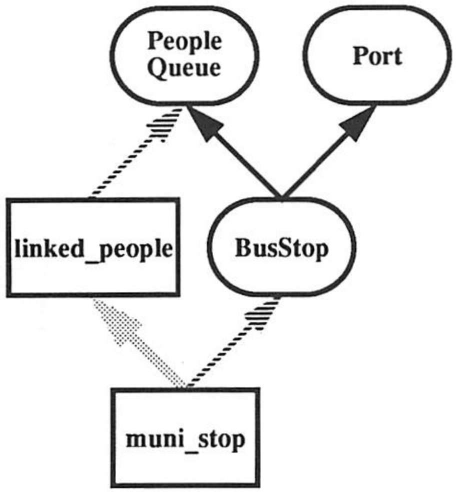

Figure 16. Resulting C++ class lattice for `muni_stop`

Figure 16 graphically represents the resulting C++ class lattice for the `muni_stop` implementation. Similarly, figure 17 graphically represents the resulting class lattice for the `inter_city` implementation. The ovals denote interfaces, the rectangles denote implementations, the solid arrows represent the *isa* relation, the dashed arrows represent the *implements* relation and the grey arrows represent the *reuses* relation. From a C++ point of view, these distinctions are irrelevant; the ovals and rectangles are all classes and the structure represents a multiple inheritance class lattice.

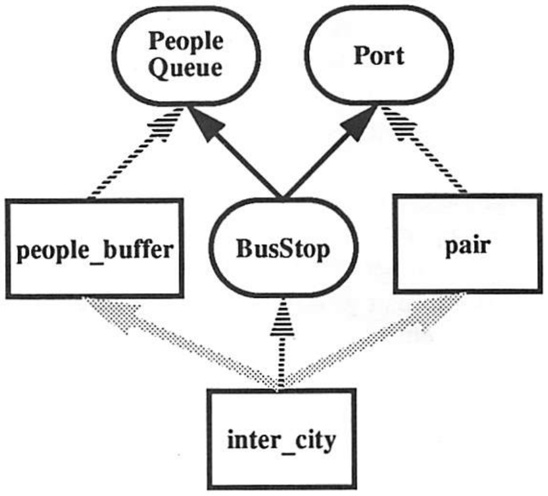

Figure 17. Resulting C++ class lattice for `inter_city`

3.4 Binding implementations to interfaces

If a class provides an implementation of an operation, the programmer declares and defines the member function for that class. The translation described above works.

On the other hand, if the implementation reuses other implementations, the obvious translation does not quite work. Consider the C++ representation of the `inter_city` implementation of figure 11 to be the C++ code of figure 18.

```
class inter_city :
      public virtual BusStop,
      public: pair,
      public: people_buffer {
   public:
      inter_city();
      virtual boolean covered() {
         return true;
      }
};
```

Figure 18. <u>Incorrect</u> C++ for the `inter_city` **implementation**

C++ inheritance does not work directly; the class in figure 18 inherits pure virtual functions and implemented functions. In C++ 2.0 inheriting a pure virtual function is not allowed. In C++ 2.1 it is allowed but the resulting class is still viewed as ambiguous.[5]

To overcome these problems, the preprocessor for the separation model generates explicit calls to the inherited functions. For example, the `inter_city` implementation given in figure 11 reuses the `enq` member function from `people_buffer`. In the generated C++ code of figure 19, an explicit call is generated to `people_buffer::enq`.

```
class inter_city :
    public virtual BusStop,
    public: pair,
    public: people_buffer {
  public:
    inter_city();
    virtual boolean covered() {
      return true;
    }
    virtual int enq(person *p0 ){
      return
       people_buffer::enq(p0);
    }
    virtual person *deq() {
      return
       people_buffer::deq();
    }
    virtual int size() {
      return
       people_buffer::size();
    }
    virtual city *dest (); {
      return pair::dest();
    }
    virtual time *departs () {
      return pair::departs();
    }
};
```

Figure 19. **<u>Correct</u> C++ code for the `inter_city` implementation**

Of course, in order to preserve C++ inheritance for implementations, calls are not generated when the multiple inheritance of implementations is ambiguous. In that case, the programmer must declare and define a member function to resolve the ambiguity.

We note that the above binding discussion does not pertain to state or local member functions since they are not part of the abstract classes representing interfaces. C++ inheritance applies directly.

Finally, to tie all of this together, we present the generated C++ class lattice in appendix A for the `muni_stop` implementation. Without the language extensions, appendix A represents what the programmer would write instead of the interfaces in figures 3 through 5 and the implementations in figures 7 and 9.

3.5 Object Layout

When an object is defined by separate interface and implementation lattices, the C++ layout of the object in memory is analogous to the logical separation model described in this paper. The layout of an object contains an implementation part and an interface part. The implementation part of an object is *physically separate* from the interface part. Furthermore, for all objects meeting the same interface, the interface part is structurally identical; the differences in representation are found only in the implementation part of the object. This is as it must be for a compiler to generate client code without knowing anything about the implementation of an object.

Figure 20 graphically depicts the layout of an object implemented by `muni_stop`. The implementation part is above the thick line and the interface part is below it. As a result of using virtual base classes to represent interfaces, the addresses of the various interfaces supported by the object are also stored as pointers in the implementation part. Type casting from the implementation to one of the supported interfaces by the object simply follows the pointer. The addresses of the contained interfaces are stored as pointers in the interface part. Type casting from an interface to one of its contained interfaces simply follows the

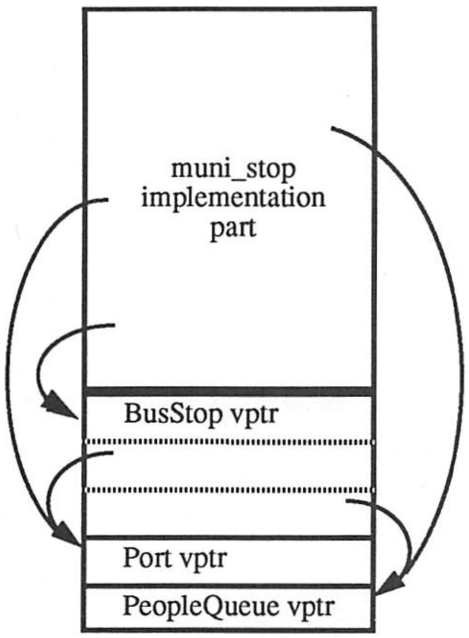

Figure 20. Physical layout of an object implemented by `muni_stop`

pointer. An `BusStop` object implemented by `inter_city` differs from figure 20 in the implementation part only. The structure of the interface part is identical.

3.6 Performance

Characterizing the execution and storage cost of using a *programming style* is tricky. What do we compare a program written according to the style to? The use of the programming style may have influenced the programmer to define and implement a completely different set of abstractions to achieve the same task.

We can, however, characterize the performance of a specific transformation. Assume we mechanically transform an existing C++ class lattice into a separate interface lattice and a single implementation lattice such that the transformed code has the same behavior as the original code. The original class lattice, the interface lattice and the implementation lattice all have the same structure.

The interface lattice contains operations defined by the public member functions of the original class lattice. For non-function public members, we introduce operations of the same name.

The implementation lattice adds the non-public members of the original C++ class lattice. The non-function public members are implemented as simple inline functions that return their values.

We compare the differences in execution time and storage use.

3.6.1 Execution cost

The original and transformed code have the same execution cost. All virtual functions in the original class lattice remain virtual in the transformed code. Since the separation model requires that all operations are virtual functions, non-virtual member functions become virtual functions in the transformed code. However, to maintain the same behavior, all calls to non-virtual functions in the original code must be transformed to class qualified calls. Because of these transformations of function calls, the original calls to non-virtual functions remain direct function calls. Similarly, inline functions are still expanded at the point of the call.

3.6.2 Storage cost

Instances in the transformed code require more space than instances in the original C++ code.

The implementation part of an instance of a transformed class is increased by a pointer to each virtual base class representing the supported interfaces. (See the arrows from the implementation part of figure 20.)

The instance also includes storage for the interface part. Each interface in the interface part stores a pointer to its virtual function table, a pointer for each of its contained interfaces and replicates its immediate contained interfaces.

4.0 Conclusions

In this paper we have reviewed the motivation for separating the interface to an object from the implementation of it. We have described how a separate interface lattice and multiple implementation lattices can be achieved in a C++ program. We have described some minor language extensions that make the separation model apparent to the C++ programmer. We have implemented a preprocessor for the language extensions.

We have described how the separation model is realized in C++. As such, the separation model could be viewed as a C++ programming style. However, C++ could support this style, without imposing it, more directly. In order to eliminate the awkward binding of implementations, described in section 3.4, pure virtual functions should be distinguished in C++ inheritance semantics. In particular, a derived class should be able to inherit a pure virtual function. Furthermore, no ambiguity should result if a pure virtual function and an implemented function of the same name are inherited; the implemented function should be inherited in favor of the pure virtual function declaration. These changes would make this a more palatable C++ programming style.

The separation model forces a programmer to create at least two names -- one for the interface and one for each implementation. The use of pure virtual functions in C++ does not. Requiring a separate name for the interface is what results in modular, flexible, distributed code.

Acknowledgments

I would like to thank Peter Canning, Mike Cannon, William Cook, Warren Harris, Walt Hill, Jussi Ketonen, Bob Shaw and Alan Snyder for commenting on the C++ separation model.

References

[1] Peter Canning, William Cook, Walt Hill and Walter Olthoff. "Interfaces for Strongly-typed Object-oriented Programming", In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 457--467, 1989. Also Technical Report STL-89-6, Hewlett-Packard Labs.

[2] David Detlefs, Maurice Herlihy, Karen Kietzke and Jeannette Wing. "Avalon/C++: C++ Extensions for Transaction-based Programming", In *USENIX C++ Workshop*, 1987.

[3] Yvon Gourhant and Marc Shapiro. "FOG/C++: A Fragmented Object Generator", In *Proceedings of the 1990 Usenix C++ Conference*, April 1990.

[4] John Interrante and Mark Linton. "Runtime Access to Type Information in C++", In *Proceedings of the 1990 Usenix C++ Conference*, April 1990.

[5] Stan Lipman. Electronic mail from Stan Lipman, ATT Bell Laboratories, April 1990.

[6] Bruce Martin, Charles Bergan, Walter Burkhard and J.F. Paris. "Experience with PARPC", In *Proceedings of the 1989 Winter USENIX Technical Conference*. Usenix Association, 1989.

[7] S. K. Shrivastava, G. N. Dixon, F. Hedayati, G. D. Parrington and S. M. Wheater. "A Technical Overview of Arjuna: a System for Reliable Distributed Computing", Technical Report 262, University of Newcastle upon Tyne, July 1988.

[8] Robert Seliger. "Extended C++", In *Proceedings of the 1990 Usenix C++ Conference*, April 1990.

[9] Alan Snyder. "Encapsulation and Inheritance in Object-oriented Programming Languages", In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages and Applications*. Association of Computing Machinery, 1986.

[10] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 1986.

[11] Bjarne Stroustrup. "Multiple inheritance for C++", In *Proceedings of the EUUG Spring 1987 Conference*, May 1987.

Appendix A: Generated C++ code for muni_stop:

```
// Interfaces:

class PeopleQueue {
  public:
    virtual int enq(person *)=0;
    virtual person *deq()=0;
    virtual int size()=0;
};

class Port {
  public:
    virtual city *dest()=0;
    virtual time *departs()=0;
};

class BusStop : public virtual PeopleQueue, public virtual Port {
  public:
    virtual boolean covered()=0;
    virtual int enq(person *)=0;
    virtual person *deq()=0;
    virtual int size()=0;
    virtual city *dest()=0;
    virtual time *departs()=0;
};



// Implementations:

class linked_people : public virtual PeopleQueue {
    class recptr *head ,*tail;
  public:
    linked_people () { head =tail =NULL ; }
    virtual int enq (person *);
    virtual person *deq ();
    virtual int size ();
};

class muni_stop : public virtual BusStop, public linked_people {
    boolean shelter ;
  public:
    muni_stop (boolean cover ) { shelter =cover;}
    virtual int enq(person *p0) {
      return linked_people::enq(p0);
    }
    virtual person *deq() { return linked_people::deq(); }
    virtual int size() { return linked_people::size(); }
    virtual city *dest();
    virtual time *departs();
    virtual boolean covered() { return shelter; }
};
```

Signature-Based Polymorphism for C++

Elana D. Granston
Center for Supercomputing Research and Development
University of Illinois at Urbana-Champaign
Urbana, IL 61801
granston@csrd.uiuc.edu

Vincent F. Russo
Department of Computer Science
Purdue University
West Lafayette, IN 47906
russo@cs.purdue.edu

Abstract

Polymorphism[CW85] is defined as the potential for a variable or function parameter to have a value of one type at one point in time, and a value of a different type at another. In this paper, we propose an extension to the C++ language that offers the polymorphism flexibility of multiple inheritance without its software engineering disadvantages. Rather than relying on inheritance, our extension allows polymorphism to be achieved by explicitly defining a *signature* (public class interface) required at a usage or call site. Any object whose class *conforms* to the designated signature can be bound to a variable or formal parameter at that site. When a member function is invoked on a pointer or reference with a signature as its declared type, the corresponding member function of the referenced object's actual class is invoked. The class's member functions that correspond to the signature's member functions can be `virtual`, `inline`, neither or both.

1 Introduction

A fundamental feature of object-oriented programming[Weg87, HO87] is *polymorphism* [CW85]. Polymorphism is the potential for a variable or function parameter to have a value of one type at one point in time, and a value of a different type at another. At present, the use of inheritance is the *only* mechanism for achieving polymorphism in C++ [ES90]. With single inheritance, the range of polymorphism that can be supported is limited. Multiple inheritance extends the range of polymorphism that can be supported, but not without incurring several software engineering disadvantages.

In this paper, we propose an extension to the C++ language that offers more polymorphism flexibility than multiple inheritance without these disadvantages. Rather than relying on inheritance, our extension allows polymorphism to be achieved by explicitly defining a *signature* (set of public member functions) required at a usage or call site. *Any* object whose class *conforms* to the designated signature can be bound to a variable or formal parameter at that site. When a member function is invoked on a pointer or reference with a signature as its declared type, the corresponding member function of the referenced object's actual class is invoked. The class's member functions that correspond to the signature's member functions can be `virtual`, `inline` neither or both.

The remainder of this paper is organized as follows. Section 2 expounds on the definition of polymorphism and discusses the disadvantages and limitations of C++ polymorphism mechanisms. Section 3 introduces our language extension. Section 4 proposes a naive implementation. Section 5 discusses the performance tradeoffs of our proposed implementation relative to existing methods. Section 6 discusses potential extensions. Finally, Section 7 draws conclusions as to the relative efficacy of the proposed extension.

2 Mechanisms for Achieving Polymorphism in C++

Single inheritance provides the simplest form of polymorphism in C++. For example, consider the following two classes:

```
class ListOfInt {
    ...
public:
    ...
    virtual void add( int x );
};

class OrderedListOfInt :  public ListOfInt {
    ...
public:
    ...
    void add( int x );
};
```

We can define a function `initialize` with a formal parameter of type `ListOfInt` as follows:

```
void
initialize( ListOfInt * aList, int n )
{
    while( int i = 0; i < n; i++ ) {
        aList->add( random() );
    };
}
```

Although the formal parameter `aList` is defined with type `ListOfInt`, the actual arguments to `initialize()` can have either type `ListOfInt` or type `OrderedListOfInt`. Hence, `initialize` is polymorphic with respect to the formal parameter `list`.

Note that since `add()` is virtual, C++ retains sufficient knowledge regarding the actual class of an argument bound to the formal parameter `list` so that it can be determined at run time whether to call `ListOfInt::add()` or `OrderedListOfInt::add()`. This run-time binding make polymorphism a very powerful mechanism in C++.

In the above example, polymorphism is supported through single inheritance. However, there are many cases where single inheritance is insufficient to achieve the desired polymorphic behavior. For example, suppose that we have two libraries containing hierarchies of classes for X-Windows display objects. One hierarchy is rooted at `OpenLookObject` and the other at `MotifObject`. All the classes in each hierarchy implement the `display()` and `move()` member functions. Further suppose that we obtained the libraries from two separate vendors and that they have only provided the header files and binaries. Can a display list of objects be constructed that can contain objects from *both* class libraries *simultaneously*? Since source code is not available for the two libraries, the implementations cannot be modified to *retroactively* inherit from a common base class.

An extremely inelegant solution involves making each element of the display list a discriminated union (i.e. union plus tag field) as follows:

```
struct displayListElement {
    int tag;
    union {
        MotifObject * pMotifObject;
        OpenLookObject * pOpenLookObject;
    };
};
...
displayListElement displayList[NELEMENTS];
```

Since an assignment to a union causes C++ to lose information about the actual type of an object, a tag must be explicitly set on assignment to indicate whether the object is from the `MotifObject` hierarchy or the `OpenLookObject` hierarchy. This information is needed to determine whether to call `displayList[i].pMotifObject->display()` or `displayList[i].pOpenLookObject->display()`. This solution violates good object-oriented programming style.

Multiple inheritance [Str87] can also be used to solve this problem. First, we construct a new class `XWindowsObject` that defines virtual functions `display()` and `move()` as follows:

```
class XWindowsObject {
public:
    virtual void display() = 0;
    virtual void move() = 0;
};
```

Next, we create a set of new classes, where each class corresponds to an existing library class. Each new class multiply inherits from `XWindowsObject` and its corresponding library class and redefines `display()` and `move()` to call the corresponding function in the library class. For example:

```
class XOpenLookCircle :  public OpenLookCircle, public XWindowsObject {
public:
    void display() { OpenLookCircle::display(); }
    void move( int x, int y ) { OpenLookCircle::move( x, y ); }
};
```

and

```
class XMotifSquare :  public MotifSquare, public XWindowsObject {
public:
    void display() { MotifSquare::display(); }
    void move( int x, int y ) { MotifSquare::move( x, y ); }
};
```

Finally, we can create the display list as follows:

```
XWindowsObject * displayList[NELEMENTS];
```

Creating these classes can be simplified by using templates [Str88, ES90]. For example given the template:

```
template<class T>
class XWindowsObjectTemplate :  public XWindowsObject, public T {
public:
    void display() { T::display(); }
    void move( int x, int y ) { T::move( x, y ); }
};
```

the necessary classes could be created as follows:

```
typedef XWindowsObjectTemplate<MotifSquare> XMotifSquare;
typedef XWindowsObjectTemplate<OpenLookCircle> XOpenLookCircle;
```

Even with templates, this option entails substantial software engineering costs. Building all these extra classes is tedious at best and muddles the program name space with a superfluous set of new classes. Despite these drawbacks, however, this is currently the *only* method for achieving polymorphism in such cases.

Note that in the above example, it is the multiple inheritance and not the templates that permit the polymorphic behavior of the new leaf classes. While templates are useful for many purposes, they *cannot* be used to achieve polymorphism. Templates allow the same body of source code (function or class implementation) to be instantiated with one of many types. They do not allow the same body of source code to be used in a single instantiation with multiple, unrelated types (no common base class). For example, templates can be used to create a generic display list class `DisplayList<`*type*`>`. parameterized by the type of elements in the list. This template can then be instantiated as either a list of `MotifObjects` (i.e. `DisplayList<MotifObject>`) or `OpenLookObjects` (i.e. `DisplayList<OpenLook-Object>`), but *not* both simultaneously. In other words, `DisplayList<OpenLookObject` *and/or* `MotifObject>` is the illegal.

The difficulty in creating a display list that can contain both `MotifObjects` and `Open-LookObjects` is that C++ constrains the type of an object reference or pointer to a class, and only provides one mechanism (inheritance and virtual functions) to achieve polymorphism of this form. Even if every class in both libraries implements the `display()` and `move()` functions with the same interface, there is no way in C++ to express this as being the *only* requirement for addition to our display list.

3 Signatures

To address some of the problems discussed in the previous section, we propose augmenting the current C++ type system to allow *signature conformance* to be specified explicitly. We define the signature of a class `X` (`sigof(X)` in our notation) to be the member functions (along with their return types and argument types) in the public interface of `X`. The signature of a class includes public functions defined by the class as well as those inherited from its base classes. Therefore, `sigof(X)` specifies the *interface provided by* class `X`, as opposed to an *instance of* class `X` (or any derived class of `X`). By conformance, we mean strict conformance. If class Y conforms to sigof(X), then for each member function in the public interface of X, there is a member function in the public interface of Y with the exact same name, return type, number of arguments, ordering of arguments, and argument types. Since there is assumed to be a canonical ordering to signature functions that does not necessarily equal the lexical order of the associated declaration, the ordering of the member function declarations in either class is irrelevant. It should be noted that a class's signature does *not* include its constructor, but does include its destructor. The only thing special about the destructor is that any destructor conforms to any other destructor. In other words, the name of the destructor (only) is ignored when checking conformance.

For example, given the two class definitions:

```
class A {
public:
        A();
        ~A();
        virtual int a();
        void b( int );
        float f;
};
```

and

```
class B {
private:
        void c( int );
        int d;
public:
        void b( int );
        int a();
        B();
        ~B();
};
```

sigof(A) is equivalent to sigof(B) and the two signatures can be used interchangably.

Only references and pointers to type sigof(X) are allowed. These have type "sigof(X) *" or "sigof(X) &". Attempting to create an instance of sigof(X) is meaningless and therefore, invalid. The address of any variable whose nominal type conforms to the signature of class X is an acceptable r-value for an assignment to a variable of type sigof(X) *. Likewise, any variable that conforms can be assigned to a sigof(X) &. By nominal type, we mean the static type (type declared in the code) of the r-value. Due to inheritance, the actual type of the object may be a derived class of the nominal type.

We also allow pointers and references with base type sigof(Y) to be assigned to those with base type sigof(X), providing Y's signature conforms to X's. If Y is a derived class of X this happens automatically since Y inherits all the members functions of X. However, this form of signature conformance is not restricted to derived classes. For example, given the two unrelated classes:

```
class W {
        ...
public:
        int a();
        void b( int );
};
```

and

```
class Z {
        ...
public:
        int a();
        char * c();
        void b( int );
};
```

the assignment

```
    sigof(Z) * aZ = ...;
    sigof(W) * aW = aZ;
```

is valid even though there is no class inheritance relationship between W and Z at all. The reverse assignment is obviously invalid.

When a member function of a signature pointer or reference is called, the implementation of that function defined by the referenced object's actual class is invoked. Our mechanism places no restrictions on the actual implementation and/or declarations of the member functions in either a class being used for its signature, or the class of an object being assigned to such a variable. The member functions can be `virtual`, `inline`, neither, or both. It is interesting to note that our technique allows variables and functions with signature arguments to behave polymorphically even when the referenced objects do not have virtual functions.

To motivate signatures further, let us return to the example introduced in Section 2. Recall that we have two class hierarchies for displaying objects on an X-Windows display. One is rooted at `MotifObject` and the other at `OpenLookObject`. Our signature proposal allows variables of type `sigof(XWindowsObject) *` or `sigof(XWindowsObject) &` to be declared, and instances of *any* class that conforms to this signature be assigned to them. Unlike multiple inheritance, our language extension allows such conformance to be inferred by the compiler. We do not require conformance to be explicitly coded by inheriting from additional classes. In particular, we could simple declare the display list as:

```
sigof(XWindowsObject) * displayList[NELEMENTS];
```

and directly assign existing classes from either library. For example:

```
displayList[1] = new MotifSquare();
displayList[2] = new OpenLookCircle();
```

We achieve the same result as the multiple inheritance solution without cluttering the program with placeholder classes like `XMotifSquare` and `XOpenLookCircle`, and without modifying existing code to retroactively inherit from a new base class. An additional disadvantage to the multiple inheritance solution lies in the implementation of multiple inheritance itself and the inefficiencies it introduces[Car90]. In particular, objects instantiated from the new leaf classes have multiple virtual function tables. Each object is increased by the space to store the extra table reference(s).

The signature based solution we propose provides this same flexibility with none of these disadvantages. Objects do not grow at all in size, no existing code needs to be modified, and no new subclasses need to be introduced.

4 Implementation

The implementation of our technique centers the representation of a signature pointer, `sigof(C) *`, or signature reference, `sigof(C) &`, as a pair (*optr*, *sptr*), in which *optr* is a pointer to the object bound to the pointer or reference and *sptr* is a pointer to a signature-specific function lookup table *stbl*.[1] When a signature reference is constructed, either as the result of an assignment or as the result of passing a conforming object to a function with a signature argument, the object's address is assigned to the *optr* field and the proper signature function lookup table is assigned to the *sptr* field. The implementation we propose uses a "thunk" technique that allows the signature table and thunk for any valid assignment to be constructed at compile time and, when necessary, filled in at link time. The *stbl* table is filled with the addresses of thunks. Each thunk indirectly invokes the corresponding function

[1]Note that this implementation precludes a signature pointer from being cast to an `int`, or other integral type, and back. We believe that this will not be a significant limitation in practice.

of the object's class after first updating the this pointer to point to the object rather than the signature reference or pointer. Next, the thunk branches to the corresponding function defined by the object's class. If necessary, multiple inheritance adjustments of the object pointer are also done in these thunks.

Calling a member function of a signature reference proceeds similarly to a normal C++ virtual function call except that the address of the function to call is obtained by indexing through the table pointed to by the *sptr* field of the signature reference or pointer, rather than indexing through the object's virtual function table (if it even has one). The thunk is called by passing the address of the signature reference as the this pointer (this indirection is stripped off by the thunk), and passing the remaining arguments as if calling a normal member function. For example, given:

```
class X {
    ...
public:
    void a( int, int );
    void b();
} anX;

sigof(X) * pX = &anX;
```

the call:

```
pX->a( 10, 20 );
```

would compile to code similar to the following:

```
(* pX.sptr[n])( &pX, 10, 20 );
```

where *n* is the index into the signature table corresponding to the member function called. These indices are assigned similarly to the way those for classes with virtual function are assigned.

All signature thunks are all basically of the form:

```
this = optr;
branch function entry point;
```

The first statement removes the level of indirection the signature reference or pointer introduces, and the second statement branches to the proper function's entry point. The code generated by the compiler is assumed to otherwise leave the stack and all registers unchanged, so when the actual class function is called, the proper arguments are supplied in the proper places.

A signature table and corresponding thunks must be constructed for each unique signature/class pair resulting from assignments in a given compilation unit. In cases where an object being assigned to a signature pointer or reference has no virtual functions, the signature function lookup table and thunks are simple to construct because the addresses of all the functions referenced via the thunks are known at compile time (or link time). Hence, no run time penalty is incurred for creating either the thunks or the *stbl* tables. For example, given the class X defined earlier, and the pointer

```
sigof(X) * pX;
```

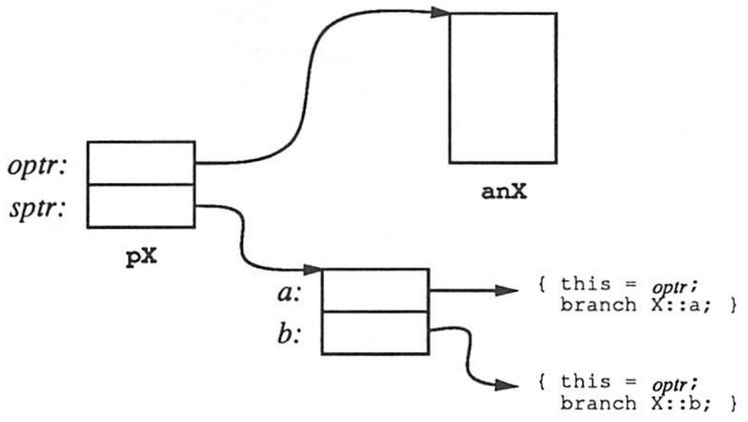

Figure 1: Result of assigning &anX to pX

the assignment

```
    pX = &anX;
```

would yield the situation shown in Figure 1. If an instance of another class meeting the X signature, say:

```
    class Y {
        ...
    public:
        int a( int, int );
        char * c( int );
        void b();
    } aY;
```

were assigned to pX, the situation in Figure 2 would result.

If the object's class defines a member function corresponding to an `inline` signature function, a non-inline version of the function is compiled and the address of this version is used in the thunk.[2] Alternately, the body of the inline function could be expanded into the thunk.

When a function is defined as virtual in the class of the r-value object, the virtual function table of the object must be indexed at run time. Such a thunk has the following structure:

```
    this = optr;
    branch *(this->vptr[n]);
```

Signature tables and thunks are generated based on the nominal type of the r-value of the assignment to a signature pointer or reference. During compilation, every assignment to a signature pointer or reference lexically encountered in a compilation unit causes a signature function table and corresponding thunks to be generated as necessary. Since a class fixes the

[2]This is similar to the case when a member function is declared to be both `virtual` and `inline` in existing C++ compilers.

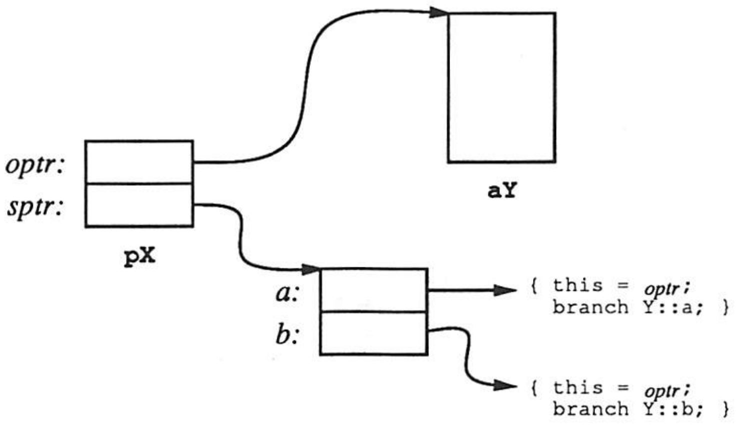

Figure 2: Result of assigning &aY to pX

indices of virtual functions into the virtual function table for all its subclasses, the nominal type of the r-value of an assignment is sufficient to both determine the offset of the virtual function table pointer from the **this** pointer and to determine the needed index into the table. As with virtual functions in C++, all the information necessary to construct the thunks and *stbl* tables is available at compile time.

For example, given:

```
class Z {
    ...
public:
    virtual void b();
    virtual void e();
    virtual int a( int, int );
    int d();
} aZ;
```

the assignment pX = &aZ would yield the situation shown in Figure 3.

The last case of interest occurs when one signature pointer or reference is assigned to another. This is only valid when the r-value signature conforms to the l-value signature. In this case, the *optr* field of the l-value is set to the r-value and the thunks referenced by the l-value's *stbl* index into the *stbl* of the r-value to obtain the next level of thunk. For example:

```
this = optr;
branch *(this->sptr[n]);
```

In other words, an extra level of indirection is performed for each level of additional signature qualification added to an object. Each higher level thunk simply strips off a level of indirection and branchs to the next level of thunk. Eventually, this chain of indirection terminates and the actual member function is reached by one of the two earlier mechanisms. In theory, this recursion is bounded by the number of levels of signature qualification which in turn is bounded by the maximum number of public member functions in any class. In

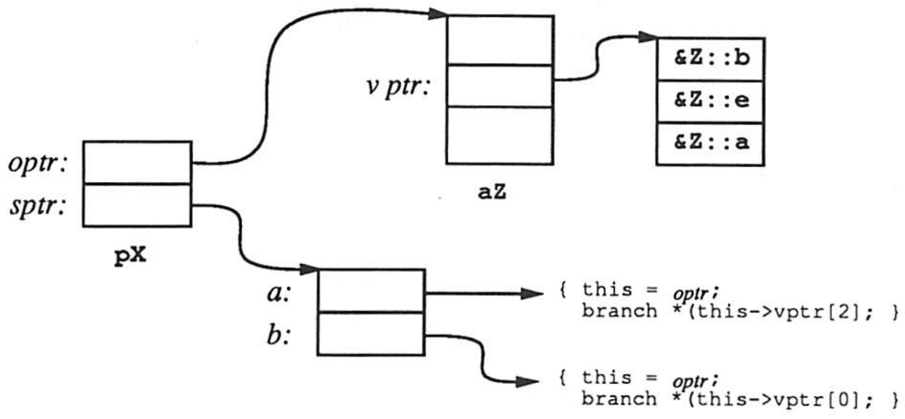

Figure 3: Result of assigning &aZ to pX

practice, the depth of this recursion is not expected to exceed one or two levels in most cases.

For example, given the declarations:

```
class U {
    ...
public:
    int a();
    int b();
};
```

and

```
class V {
    ...
public:
    int a();
    virtual int b();
    int c();
};
```

the statements:

```
V aV;
sigof(U) * pU;
sigof(V) * pV;
...
pV = &aV;
pU = pV;
```

would yield the situation show in Figure 4.

5 Performance

The space cost of our proposed implementation is two-fold. First, it involves the storage necessary to hold the signature function tables and thunks. The size of each table is one word per signature member function. Likewise, there is one compiled thunk per signature function. The size of each thunk is instruction set dependent. For most architectures, the extra dereference to obtain the proper object pointer is likely to compile to a single instruction. The remaining logical instruction in the thunk is the branch to the member function's entry point. In cases where the function being called is not virtual with respect to the object's actual class, this is a direct branch that is likely to compile to a single

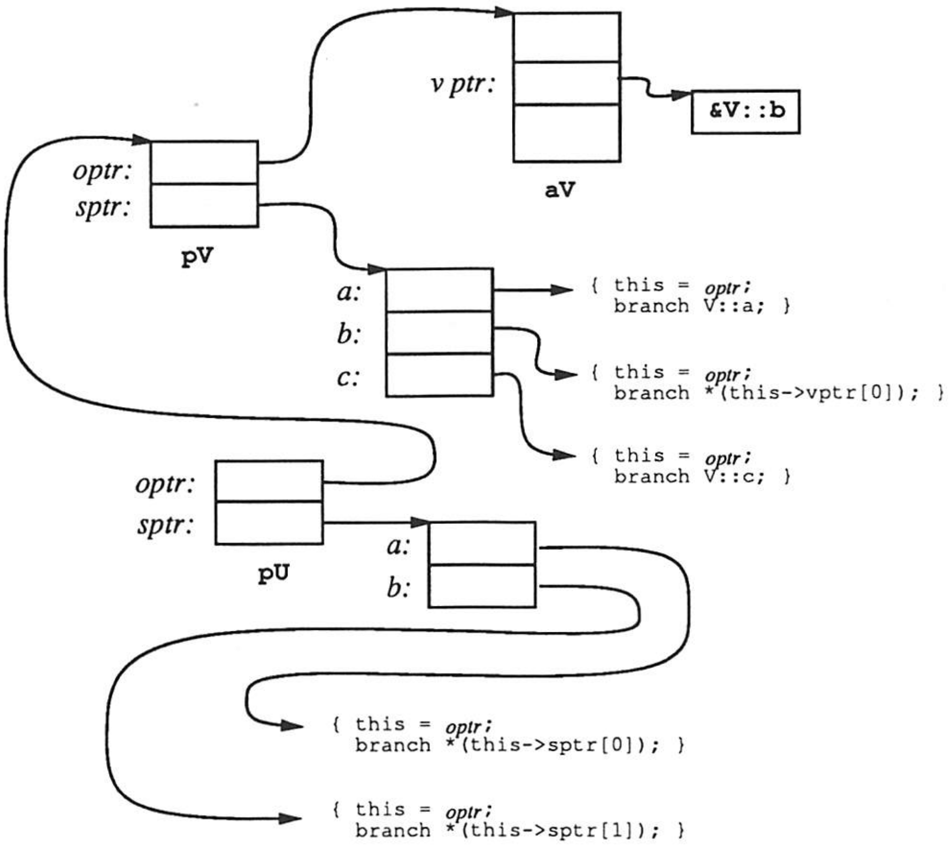
v ptr:
&V::b
aV
optr:
sptr:
pV
a:
b:
c:
{ this = optr; branch V::a; }
{ this = optr; branch *(this->vptr[0]); }
{ this = optr; branch V::c; }
optr:
sptr:
pU
a:
b:
{ this = optr; branch *(this->sptr[0]); }
{ this = optr; branch *(this->sptr[1]); }

Figure 4: Result of assigning pV to pU

instruction. The other kind of thunk occurs when the thunk either indexes an object's virtual function table, or indexes the *stbl* table of another signature pointer or reference. In these cases the generated code must dereference the signature or object pointer to obtain a function table address, index this table, and the branch to the function. This should compile down to approximately one instruction to obtain the table pointer, at most a couple instructions to index the table, and finally one instruction to perform the branch. In cases where the thunks must perform multiple inheritance adjustments of the `this` pointer for an object, this cost must be added as well.

Tables and thunks can sometimes be shared. In particular, if there are multiple assignments where the nominal type of the r-value and signature of the l-value are the same, the same table and thunks can be used at each point. Likewise, most techniques that optimize the number of virtual function tables generated by a compiler across compilation units can be applied to the signature tables and thunks as well.

The second space cost of this implementation results from that fact that signature pointers and references occupy an additional word of storage over normal C++ pointers and references. This extra word is used to hold the pointer to the signature function table. While it does have the disadvantage of not allowing a signature pointer to be cast to an integer and back, the additional word of storage does not seem prohibitive.

The run time penalty imposed by this solution has three-fold. First, binding an object to a signature pointer or reference involves the writing of an additional field. Second, the copying a signature variable to another of the same type involves two reads and two writes as opposed to the single read and write required to copy a normal pointer. This should not be prohibitive in practice.

The third cost is associated with the calling of member functions through signature pointers or references. At the call site, the cost of our implementation is equivalent to the cost of invoking a virtual function in non-multiple inheritance C++ compilers. Namely, an additional pointer dereference and a table lookup are required to obtain the function address (branch target). This cost is paid for *all* calls. Functions defined as non-virtual, or inline in the actual class of the object, will cost the same as virtual functions would at this point.

The additional costs are incurred suring execution of the thunks. These depend on whether the actual function being called is virtual, non-virtual, or inline. All thunks incur the extra cost of the additional pointer dereference to obtain the proper object pointer before branching to the eventual function entry point. On a pipelined architecture, this dereference can be scheduled in the delay slot of the branch.

When the function being invoked is non-virtual, the only additional cost is a direct branch to the function's entry point. If the function is inline, the cost is the same if there is a non-inline version of the function compiled, or the cost is zero if the function is inline expanded into the thunk. When the function is virtual, the additional cost involves indexing the object's virtual function table to obtain the entry point of the proper function. When multiple levels of signature qualification are present, this cost is paid for each level of qualification. Since the ordering of functions in a signature is unimportant, additional qualifications must each be to a signature with at least one less function than the previous one. Therefore, as mentioned in Section 4, the theoretical bound on the number of such indirections is the number of public member functions in the original class, though the number of indirections in practice is expected to be one or two, in most cases.

It is important to note that these costs are only incurred at points where signatures are used. There are neither tables nor thunks in compilation units that do not contain assignments to signature pointers or references, normal C++ pointers and references are unaffected, and normal C++ function calls are unchanged.

6 Possible Extensions

Three possible extensions to our proposed language feature are:

- extending signatures to include public *data* members,
- allowing aliasing of the names in a signature,
- allowing the bulitin types to conform to certain signatures.

6.1 Extending Signatures to Include Data Members

We could extend `sigof(X)` to include not only the function members in the public interface of `X` but *also* any *data* members in the public interface of `X`. Implementing this would be straightforward and likely involve either storing offsets to access the given variable in the *stbl* table for a signature pointer or reference, or providing thunks to access the variables. This latter case would likely be necessary for situations where the data is present via a virtual base class. It is interesting to note that the latter case reduces to the same code as if the programmer had supplied functions to access the variables. The usefulness of allowing data in signatures seems somewhat dubious since it seems to confuse implementation with interface. Good programming practice stresses separating these two.

6.2 Aliasing of Signature Names

Suppose that in the examples in Section 3, `MotifObject` named the function to display the object `show()`, rather than `display()`. In this case, `MotifObject` would no longer conform to `sigof(XWindowsObject)`, even though it logically has all the proper operations. For this reason, it might be desirable to extend C++ syntax to allow member aliases to be specified. For example, `XWindowsObject` could be defined as:

```
class XWindowsObject {
public:
    virtual void display();
    virtual void move( int, int );

    alias show display;
};
```

This would be interpreted to mean that given `window` with type `XWindowsObject *`, the calls `window->display()` and `window->show()` are equivalent operations. We could then extend signature conformance to mean that any class with either `move` and `display` or `move` and `show` in its interface could conform to `sigof(XWindowsObject)`.

6.3 Including the Builtin Types

If the builtin types (`int, char, short`) are considered to have signatures as well, they may conform to certain other signatures. For example, the builtin type `int` logically conforms to `sigof(Counter)` if `Counter` is defined as:

```
class Counter {
public:
    operator ++();
    operator --();
    operator int();
};
```

Other classes implementing the Counter signature can also be constructed. For example, AtomicCounter could implement a counter in a multiprocessor environment, or Range-CheckedCounter could check that a counter never exceeds a maximum value. If a variable with type sigof(Counter) * or sigof(Counter) & is used in a piece of code, instances of AtomicCounter, RangeCheckedCounter, or and ordinary int could be assigned to the variable. The implementation of this would involve more sophisticated thunks that contain code to perform the operations on the primitive types directly, since corresponding member functions do not exist.

7 Conclusion

We have discussed the limitations of existing techniques for supporting polymorphism and code reuse in C++. We have proposed a new language extension that allows polymorphism to exist without relying on inheritance. We have outlined a naive implementation of our technique, and analyzed its performance. May optimizations are possible but are beyond the scope of this paper.

Our proposal allows polymorphism which can only be achieved in existing C++ by using (multiple) inheritance and modifying existing code or introducing new leaf classes. We consider this to be a major software engineering disadvantage. Due to its limited overhead, we believe that our technique has significant advantages over multiple inheritance solutions for supporting polymorphism in C++.

An implementation of this extension in the GNU C++ compiler is under investigation. We note that this paper presents a "blue sky" language extension to C++. Further evaluation of its usefulness in practice is necessary. We solicit comments as to the perceived usefulness of our technique.

8 Acknowledgements

We would like to thank Doug Lea for his insightful comments on an earlier version of this paper. We would also like to thank Ralph Johnson for helping us clarify some of the terminology used in this paper, and (unbeknown to him) for inspiring the original idea for this work through his work on type checking Smalltalk. Other inspiration for this work comes from the Emerald[JLHB87] programming language.

Elana Granston is supported by the Department of Energy under Grant No. DE-FG02-85ER25001 and Cray Research Inc.. Vincent Russo is supported by grants from Hewlett-Packard Laboratories and the Purdue Research Foundation.

References

[Car90] T. A. Cargill. Does C++ Really Need Multiple Inheritance? In *1990 USENIX C++ Conference*, pages 315–323, 1990.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, Reading, Massachusetts, 1990.

[HO87] Daniel C. Halbert and Patrick D. O'Brien. Using Types and Inheritance in Object-Oriented Programming. *IEEE Software*, pages 71–79, September 1987.

[JLHB87] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. Fine-Grained Mobility in the Emerald System. In *Proceedings of the Eleventh ACM Symposium on Operating System Principles*, pages 62–74, November 1987.

[Str87] Bjarne Stroustrup. Multiple Inheritance for C++. In *EUUG Spring '87 Conference*, 1987.

[Str88] Bjarne Stroustrup. Parameterized Types for C++. In *1988 USENIX C++ Conference*, pages 1–18, 1988.

[Weg87] Peter Wegner. Dimensions of Object-Based Language Design. In *Proceedings of the 1987 OOPSLA Conference*, pages 168–182, 1987.

Panel: How Useful is Multiple Inheritance in C++?

Moderator: Andrew Koenig
Panelists: Tom Cargill, Keith Gorlen, Rob Murray, Mike Vilot

Panelists' Statements

Tom Cargill
Consultant
P.O. Box 69
Louisville, Colorado 80027

C++ is a programming language. Multiple inheritance (MI) is useful in C++ if better programs can be written with MI than without it. The usefulness of MI in C++ should be determined by writing programs with MI and comparing them with corresponding non–MI programs. Of course, whether one program is ''better'' than another is a complex question. There are few objective measures of properties such as simplicity and readability, and so the evaluation must be largely subjective.

My evaluation of the evidence to date is that MI is not useful. I have examined all the published examples of MI and found that the programs can be rewritten without MI. In most cases, MI can be replaced by simple aggregation of a member object. MI is removed without detriment to the programs; the non–MI versions are as simple and efficient as the MI versions. MI does no appear to increase the expressive power of C++; it does not improve our programs.

Research in programming languages must involve experiments in writing programs. Research on MI in C++ has focused on obtaining a consistent definition and efficient implementations. There has been little research to investigate how to use MI in writing programs. The first working implementations of MI were closely followed by compiler products. A research cycle to weigh the costs of MI against its benefits has not been undertaken. I question the diligence of the C++ technical community in conducting our research.

The issue would be less important if MI were a trivial part of the language. However, its complexity, particularly virtual base classes, makes C++ harder to learn and use. If multiple inheritance is a mistake, programmers may have to pay the price of using an unnecessarily complicated language for years to come.

Keith Gorlen
Building 12A, Room 2033
National Institutes of Health
Bethesda, Maryland 20892

Is multiple inheritance (MI) a useful feature for object–oriented programming in C++? I would give a qualified ''yes'' to this question. MI appears useful in at least three situations:

1. Decoupling a class's specification (the set of functions that may be applied to an instance) from its implementation, as suggested by Kennedy, Martin, Meyers, Vilot, and others. Public inheritance from one or more abstract base classes defines the specification, while private inheritance from one or more implementation classes enhances code sharing. While composition may frequently be used in place of private inheritance, the latter is a more powerful technique, since a derived class can define virtual functions called by the private base class to customize its behavior.

2. Modularizing the design of base classes. With single inheritance, designing the set of member functions a base class is to provide is difficult. If a function is not included in a base class, then one must occasionally test the type of a derived object at run–time and perform a downward cast in order to apply the omitted function. If a function is included in a base class, then one occasionally discovers a derived class for which the function cannot be sensibly implemented, so it is implemented to cause an error if called. This is just another way of subverting C++ static type checking.

A large base class can be decomposed into several smaller base classes, each declaring only those functions relating to a particular feature. Derived classes can then use MI to selectively inherit from one or more of these base classes, rather than being forced to inherit all the functionality of a single large base class. I have been experimenting with this technique in the context of the NIH Class Library, decomposing class Object, for example, into classes Copyable, Ordered, Scanable, Storeable, and Typed.

3. Combining independently developed class libraries in a single application program. One can easily imagine an application program that stores and retrieves objects using an OODBMS, and displays them using a GUI toolkit. The OODBMS and GUI class libraries are likely to be separately developed, yet each may expect to deal with objects that are instances of classes derived from one of its own base classes.

My "yes" is qualified because the cost of using MI is high. To obtain an indication of some of this cost, I compiled the NIH Class Library both with and without its optional MI support, and compared the code sizes of the programs in its test suite and the sizes of library objects. Code size (text + initialized data) with MI support was an average of 12% larger than without (N=36, std. dev. = 3%), and the average object size was 55% larger with MI support than without (N=33, std. dev. = 22%). Note that the NIH Class Library doesn't actually use MI, it only supports it by making class Object a virtual base class and including code to deal with virtual derivation from NIH library classes, so these numbers may be uncharacteristically low.

I also measured the extra overhead of calling a virtual function of a virtual base class compared to that of calling an ordinary virtual function and found it to be 50% greater (0.93 vs. 0.61 microseconds on a SPARCstation 1+).

Another cost of MI is the complexity it adds to C++, making it more difficult to learn and implement. MI introduces elaborate rules for base class initialization and binding of virtual function calls, and virtual base classes are tricky to use. The first C++ compiler to implement MI was released over 20 months ago, and I still do not have a version that can correctly compile one of my example programs.

Advances in computer technology will, in time, ameliorate most of the problems with MI, and it will become an essential feature of the C++ language.

Rob Murray
AT&T Bell Laboratories
184 Liberty Corner Road
Warren, NJ 07059

The value of multiple inheritance (MI) in C++ has been challenged by Cargill[1] and Schwarz[2] by pointing out the lack of published examples of real use of MI in applications, and by showing how contrived examples of MI that have appeared could be implemented using single inheritance (SI). I claim that the lack of published MI examples is due to the fact that it is simply too early for large software projects (where MI has its greatest utility) to have reached the point where they are ready to publish results.

One of the most fundamental ideas of inheritance is the collection of concepts that are common to two or more kinds of objects into a single place. In this sense, a base class specifies a set of classes that share certain common concepts and operations (by being derived from the base class that declares them). This is a powerful simplification and is directly supported by the 2.1 features of pure virtual functions and abstract base classes (ABCs).

What happens when there are two or more such sets of classes in an applications, and their intersection is not empty? The most straightforward way to express that a single object supports the concepts and operations of two or more such sets of classes is to derive from two or more base classes:

```
class Police_car : public Vehicle, public Municipal_property {
```

Here, I would like to pass Police_car*sto functions that take Vehicle*sand to functions that take Municipal_property*s. MI is the natural, obvious way to do this.

Implementing this with SI is possible but confusing. Using composition is the wrong abstraction: a Police_car*is–a*Vehicle, it doesn't *have–a*Vehicle. The use of implicit conversions implied by the use of composition is dangerous[4]; and the sharing of concepts between the two base classes (automatically handled by MI's virtual base classes) requires explicit manual support when using SI. The fact that many MI examples can be somehow implemented using single inheritance is a red herring,as is the "Cargill/Schwarz criterion"[3]: almost all C++ features can be implemented using only the C parts of the language. This doesn't mean that we should abolish virtual functions! The point is that MI is the natural way for large projects to use ABCs. If we restrict the use of ABCs to projects where the sets of classes specified by those ABCs never intersect, then projects where they *do*intersect will have to develop complicated schemes to transform their MI designs into SI implementations[2].

How expensive is MI for those who do not use it? There certainly is an expense in the sense that compiler developers spend time developing and maintaining MI when they could otherwise be working on something else. Also, the language is made larger and therefore harder to learn. However, there needn't be a compile–time or run–time penalty—I'm not aware of any compiler vendors who have claimed that their compilers would be much faster or produce better SI code if MI did not exist.

Documented uses of MI are just beginning to appear[3]. Why haven't more papers describing MI use in real software projects been published? Most C++ users are still essentially beginners, and are justifiably reluctant to use MI on their first (or even their second) design. C++ 2.0 (the first release to support MI) became available to most users around late 1989 to early 1990, with 2.1 (pure virtuals and abstract base classes) becoming available around the middle of 1990. The technology to support MI, especially using pure virtual functions and abstract base classes, hasn't been in the hands of users long enough for many significantly sized projects to have been designed using MI, developed, delivered, written up, and published.

In the panel I will talk about some of the ways I have used MI in my work, and why its use has been much more natural and straightforward than the SI alternatives I considered.

References

[1] Cargill, T. ''We must debate multiple inheritance'', *The C++ Journal*, pp. 20–22, Fall 1990.

[2] Schwarz, J. ''A Critique of the Skiplist/Associative Array Example't make programming easier'', *The C++ Report*, April 1991.

[3] Carroll, M., ''Using Multiple Inheritance in C++ to Implement Abstract Data Types'', *The C++ Report*, April 1991.

[4] Murray, R. ''Building well–behaved type relationships in C++'', *Proceedings of the 1988 USENIX C++ Conference.*

A Copying Collector for C++

Daniel R. Edelson
daniel@cis.ucsc.edu

Ira Pohl
pohl@cis.ucsc.edu

Baskin Center for Computer and Information Sciences
University of California
Santa Cruz, CA 95064

27 February 1991

Abstract

Garbage collection is an extremely useful programming language feature that is currently absent from C++. The benefits from garbage collection include convenience and safety because the programmer is not responsible for freeing dynamically allocated storage. Many reclamation schemes improve efficiency by compacting objects in memory improving locality and reducing paging. Some reclamation techniques are more efficient than manual reclamation for important classes of data structures.

This paper presents a copying collector for C++ that supports polymorphism and does not require indirection through an "object table." This memory reclamation scheme is one of few that is philosophically consistent with the design goals of the C++ programming language: one must not be penalized for features that are not used. This report includes performance measurements from a prototype implementation.

Introduction

Garbage collection (GC) is a programming environment feature that removes the programmer's responsibility for freeing dynamically allocated storage. It is a fundamental component of a Lisp or a Smalltalk-80 system. Garbage collection is also provided in some imperative object-oriented languages such as Eiffel [Mey88] and Modula-3 [CDG+88].

There is widespread belief in the C++ community that GC is useful and beneficial. While there are superficially valid reasons why GC should not be part of C++, overall, it is a desirable feature [EP90]. However, there is no consensus as to what kind of garbage collection is most appropriate. The alternatives include conservative collection [BW88], partially conservative collection [Bar88,Bar89], and non-conservative copying or mark-and-sweep collection [Ede90].

In this paper we examine the alternatives and present a copying collector. Section one explains why garbage collection in C++ is a difficult problem. Section two considers related work. Section three introduces a copying collector. Section four discusses the efficiency of the system, and section five concludes the paper with advantages and limitations of the system.

1 The Difficulty with GC in C++

1.1 How Garbage Collection Works

A garbage collector scans the program's global state searching for pointers to dynamically allocated objects. This entails examining the stack, the registers, and the global data. Each time the collector locates a pointer, it traces out the data structure reachable from the pointer. Any object that is reachable from some pointer is *live*. All unreachable dynamically allocated objects are garbage and should be deallocated.

In mark-and-sweep garbage collection each object is marked as it is visited. After all accessible objects have been marked, a *sweep* phase visits all the objects and deallocates the unmarked ones. In copying garbage collection [Bak78,LH83,FY69,Ung84], when an object is visited, it is copied to a new memory space. After the traversal the old memory space is recycled *en masse*.

Garbage collectors face two main problems: locating every global pointer, and, locating every pointer from one object to another.

1.2 Alternatives

Traditional garbage collection schemes use tagged pointers. When the collector examines a value (on the stack, for example), a tag indicates whether the value is a pointer or an integer. Pointers inside objects are also tagged. Thus, this solves both of the problems that we identified in the previous section. Unfortunately for GC proponents, tags are inconsistent with the C++ philosophy. Tagged integers have reduced range and lower efficiency, or else require hardware support. Data in the standard libraries must be tagged, reducing efficiency and penalizing all users of the language.

Alternatively, objects may be allocated from segregated memory pools, with integers, themselves, allocated from one of the pools. In this model everything is a pointer. Pointers to collected objects are precisely identifiable by their values. The added level of indirection reduces the efficiency of integer arithmetic and is another unacceptable solution.

An object table makes pointers precisely identifiable because exactly all the direct pointers are in the table. Pointers that the programmer manipulates are indirect through the object table. Objects' motion does not affect the application. This solution adds a level of indirection to every pointer dereference. Ungar found that eliminating the object table from a Smalltalk system improved efficiency dramatically [Ung86]. It is unlikely that an object table-based implementation of garbage collection would yield satisfactory efficiency.

Another alternative is conservative collection. This is a technique that does not need to differentiate between pointers and integers. Any quantity that might be a pointer, is assumed to be a pointer. Conservative collectors cannot normally move objects. This type of collection is described in §2.1.

The last choice we discuss is garbage collection by pointer tracking. This forms the basis for our collector and is described in §3.

2 Related Work

2.1 Conservative Collection

The simplest way to accomplish automatic storage reclamation in general C++ programs is to use conservative garbage collection [BW88]. Conservative GC requires virtually no compiler support. It is a variation on traditional mark-and-sweep storage reclamation that does not differentiate between pointers and integers.

When an allocation request cannot be satisfied due to insufficient free storage, the memory allocator invokes a reclamation pass. In the *mark* or *trace* phase, the collector examines the stack, the registers, and global data searching for values that might refer to dynamically allocated objects. Any such value may be either a pointer, an integer, or some other type of data, but it is interpreted as a pointer. These perceived pointers constitute the *roots* of the garbage collection. The objects they reference are marked. Each marked object is itself conservatively scanned for possible pointers. Perceived pointers are followed and objects are marked until there are no more objects or pointers to examine. The marked objects are a superset of the actual reachable data.

After the mark phase the collector examines every dynamically allocated object. Each unmarked object is deallocated. Memory allocators that work with mark-and-sweep collection add a header and footer to every allocated object containing the actual size of the memory block. Thus, given a pointer to a dynamically allocated object, the allocator can determine the size of its memory block [Knu73].

The advantage of conservative collection is that it does not require compiler assistance. However, if there are any integers that look like pointers it retains inaccessible storage. Conservative collection precludes compaction since pointers cannot be altered.

2.2 Partially Conservative Garbage Collection

Bartlett describes a garbage collector that is conservative on the stack and copying in the free-store [Bar88]. The collector was intended primarily for Scheme, but has been adapted for use with C++ programs. The allocator allocates memory from consecutive, uniform size, chunks of memory called *pages*. This page size is unrelated to the hardware page size. Pages are assigned to spaces; a page's space is indicated by an associated space identifier. There are two special spaces, *current_space* and *next_space*; they fulfill the roles of from-space and to-space in the standard copying collector algorithm. That is, a collection moves live objects from current_space to next_space, and then sets current_space equal to next_space. Any page whose space is not equal to either current_space or next_space is currently free.

The algorithm garbage collects after half of the free-store pages are used. Next_space is set to current_space plus one modulo a large number. All living objects will be copied to pages labeled with the next_space value.

Initially, the collector conservatively scans the stack, the registers, and, if appropriate, global data, looking for roots. As in other conservative garbage collection algorithms, it assumes that any value on the stack that might be a pointer is actually a pointer. The referent of each root cannot be moved, because the root might be an integer. These objects are "moved" from current_space to next_space through having the space identifiers associated with their pages set to the next_space value.

After finding roots, the collector scans the promoted objects looking for pointers to objects still in current_space. This scan is not conservative. The Scheme collector uses tags to identify the pointers; the C++ collector requires that the programmer provide a function to locate the pointers. Every object that is still in current_space is compactly copied to next_space. A forwarding pointer is left in the old copy and the pointer that led to the object is updated with the new address. Garbage collection is complete when all the objects in next_space have been scanned for pointers into current_space. When the algorithm has moved or copied all the living objects it sets current_space to next_space and resumes the interrupted allocation request.

The collector was originally non-generational and intended for use with Scheme; Bartlett later added generations and C++ support [Bar89]. Generations are implemented in the page counters. To collect C++ objects, the user provides a function to identify internal pointers. During a collection the collector calls this function to get the offsets of internal pointers. It then traverses the internal pointers and copies the objects.

As described in [Bar89] the collector does not consider the fact that a pointer may point at a derived class object. Therefore, it does not support C++'s flavor of polymorphism.

3 A Copying Collector

Two problems in implementing GC in C++ are: identifying global pointers, and, identifying pointers within objects. C++'s flavor of polymorphism compounds these problems by allowing a pointer of particular static type to point at objects of different dynamic types. Any complete solution must address and solve these problems in the context of polymorphic type hierarchies.

The scheme described in this section is compatible with existing code and libraries, efficient, encapsulated, and fully consistent with C++'s flavor of polymorphism. It is *type accurate* [HD90], meaning that a value it interprets as a pointer is statically typed to be a pointer. This contrasts with conservative collection which interprets integers as pointers. This stop-and-copy collector has the following characteristics.

- The system is modular, encapsulated and very flexible. The application may communicate with the memory manager to configure it appropriately. Any number of collectors and allocators can exist concurrently in an application on disjoint data structures.
- The allocator is fast. Allocation requests compile inline to 7 VAX instructions of which 4 are executed in the common case. It supports discontiguous chunks.
- The collector is a copying collector that collects and compacts in one pass.
- The components are small, simple and efficient. In many cases it is more efficient than a standard manual memory allocator such as the global `new` and `delete` operators or *malloc* and *free*.

A prototype of the collector has been implemented in application code outside of the compiler. An implementation within a compiler is underway. The system consists of the following components: a memory allocator; GC-related members of collected types, for example, overloaded `new` and `delete` operators, and; *smart pointer* [Str87] types.

3.1 The Memory Allocator

The strategy used in this allocator is based on the fast block allocation scheme described in [App87]. It uses a variation that supports discontiguous spaces. A *chunk* is a large block with which the allocator satisfies allocation requests. A *space* is a linked-list of chunks. The active space, that is, the one currently satisfying allocation requests, is known as *to-space.* When a chunk is exhausted the allocator may either begin a collection, or obtain a new chunk and continue allocating. Which one the allocator chooses is a policy decision that is outside the scope of this paper.

The allocator is encapsulated in a C++ `class` called a `mallor`. The common term `malloc` was not used to preclude confusion with the ANSI C [Ame89] library routine of that name. In this discussion the noun *an allocator* denotes an instance of `class mallor`.

Individual objects are not individually freed, therefore dynamically allocated blocks do not requite headers and footers. An entire space is deallocated by returning its chunks to the low-level allocator.

An allocator is constructed with a function pointer to a collection routine. When it runs out of space it takes the following steps:

1. *flip*, i.e., begin allocating from a new space,
2. invoke a garbage collection by calling through the `gc` function pointer, upon return,
3. free the old space, and
4. resume the interrupted allocation request.

This is transparent to the application programmer.

An allocator is a `static` member of the base class of the collected data structure. The overloaded `new` operator of the class obtains memory from the `mallor`.

3.1.1 Implementation

The allocator has two implementations that differ in how they detect that the current chunk is exhausted. One performs an explicit bound check during each allocation. The other uses virtual memory protection as suggested by Appel [App87] to avoid the explicit test. It write-protects the last page of the chunk and allocates until a write-fault occurs. It writes to each object to force the fault. The two versions of the allocator are respectively called the *testing* version and the *faulting* version. Experiments reported in [Ede90] show that in many cases write-faulting is not efficient enough to justify its added complexity.

3.2 Locating Pointers

A "root" of the data structure is a pointer on the stack, in a register, or in global data. This collector uses auxiliary data structures to track the roots.

In the prototype implementation, roots of a data structure are `class` objects of a parameterized type that behave as *smart pointers* [Str87]. For nodes of type *T*, roots are of type `R_`*T*. For example, the programmer does not currently manipulate local variables of type `node *`; objects of type `R_node` are used instead. When the collector is implemented in the compiler the programmer will appear to use normal pointers: smart pointers will be generated instead.

The root classes are currently generated with parameterized `#define` macros because templates [ES90] are not available. When the collector is in a compiler neither macros nor templates will be required. In the remainder of this paper, *roots* are these parameterized class objects. A root can be converted to a normal pointer type. It can be dereferenced like a normal pointer. Roots have constructors and destructors that track their lifetimes.

The roots of the collection are located using two data structures. Most roots are either global or `static` variables, or `auto` variables. These roots can be tracked with a stack. When such a root is created its address is pushed onto a stack; when it is destroyed its address is popped. Other roots, in particular those contained in other dynamic objects, cannot be tracked using a stack. Their addresses are kept in a doubly linked list.

There will generally be multiple stacks (and doubly-linked lists) created, one for each class in the polymorphic type hierarchy. However, that fact is not important in this discussion and is omitted for clarity.

3.2.1 Stackable Roots

When a global root is created, or a root is allocated on the runtime stack, the root's address is pushed onto a stack. All the global and local roots in a program can be found by traversing the stack. The stack was implemented two ways: with linked lists and with arrays.

In the array implementation a separate array of root pointers tracks the addresses of all the roots. Root construction and destruction push and pop addresses from the arrays, respectively. In the linked-list implementation, the list is threaded in the runtime stack, or in global data for non-auto roots. Each root consists of two words, its pointer and its link. A representative runtime organization is shown in figure 1. This figure presents a single root stack, implemented as a linked list, and no doubly-linked roots.

3.2.2 Doubly-Linked Roots

Dynamically allocated roots that are not in the data structure may not have LIFO lifetimes, therefore they cannot be tracked with a stack. This collector tracks non-stackable (foreign) roots with a doubly-linked list. The doubly linked list permits deletion of any list element to support non-last-in first-out (LIFO) insertion and removal.

These roots must have a distinct type. Dynamic roots to objects of type *T* are of type `DR_`*T*. They are called *droots* because they insert themselves into a doubly linked list when they are created (doubly-linked roots.) As with stacked-roots there is one list for every kind of root. The collector can find all the lists.

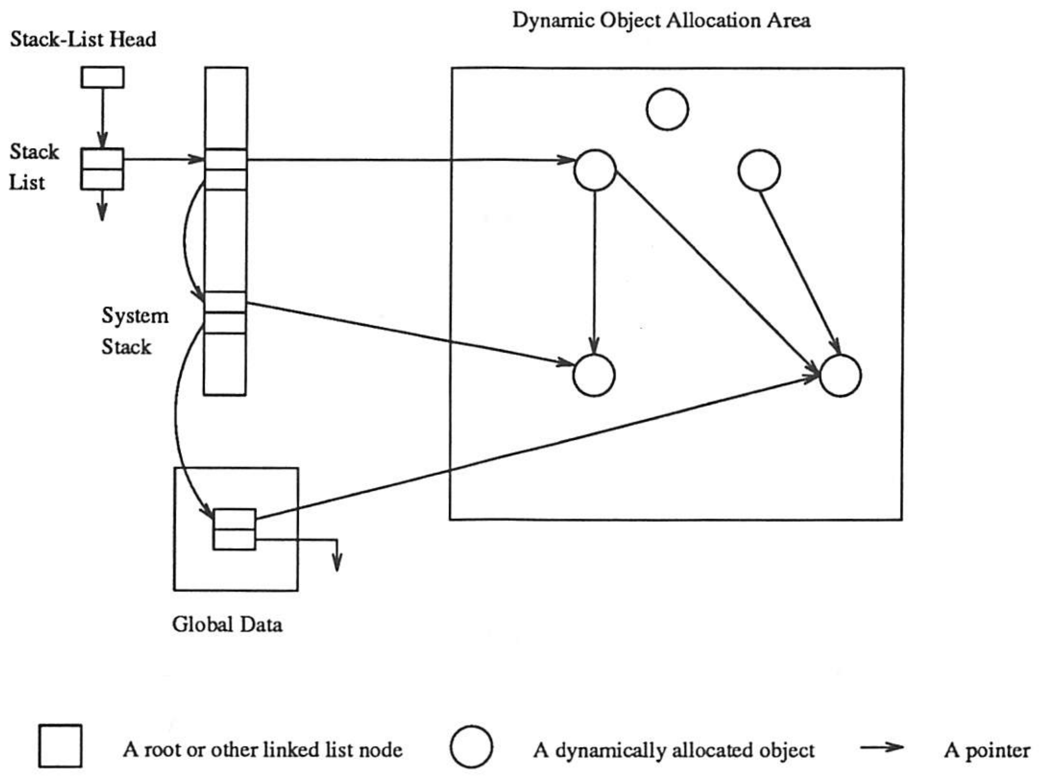
Stack-List Head
Dynamic Object Allocation Area
Stack
List
System
Stack
Global Data
A root or other linked list node
A dynamically allocated object
A pointer

Figure 1: Runtime organization of the copying collector.

The linked-list implementation of the root-stack is shown.
No doubly-linked roots are shown.

3.3 Copying

The current implementation requires that every collected type have a `copycollect()` virtual function. This function copies the object and stores a forwarding pointer in the old copy. Then, `copycollect()` updates the pointer that led to the object with the object's address and recursively copies the descendents. A sample function is shown in figure 2. The explicit type conversion required by this function is both safe and necessary; the reasons are explained in [Ede90].

When an object is copied the pointer that led to the object must be updated with the new address. When the object is of a derived class type, the pointer requires conversion from *derived* $*$ to *base* $*$. This conversion, in the presence of multiple inheritance, may change the value of the pointer.

To preserve type-safety, i.e., ensure that the conversion is safe, `copycollect` is overloaded based on the type of the pointer that references the object. This ensures that

```
class node {
    node * left, * right;
    ...
    virtual void copycollect(node * &);
};

void node::copycollect(register node * & nodep)
{
    if (nodep = (node*) get_forward()) return;      // already done?

    nodep = new node(*this);                        // copy object
    set_forward(nodep);                             // set forward ptr
    if (left) left->copycollect(nodep->left);       // copy children
    if (right) right->copycollect(nodep->right);    // copy children
}
```

Figure 2: A copy function for a node type.

enough static type information is available for correct pointer conversion. A derived class must have a copy function for every type of pointer that can lead to such an object. A simple derived class based on the node class of figure 2 is shown with its copycollect functions in figure 3.

3.4 A Collection

The collector is nonincremental and copying. It is invoked by the allocator when an allocation request cannot be satisfied. The collector traverses the root stacks and "visits" each root. From each root, it initiates a depth-first copying collection with a call to copycollect through the root. After it has collected, it returns control to the allocator.

3.5 Using the Collector

Under this scheme the attribute *collected* is a characteristic of an object. This contrasts with languages such as Modula-3 [CDG$^+$88] in which *tracked* is an attribute of a pointer.

Figure 4 shows the necessary members of a garbage collected type in the prototype implementation.

Figure 5 demonstrates the definition of a garbage collected type. After this, the programmer does not manipulate "node *" pointers. Instead, smart pointers of type R_node are used. The final item coded by the programmer is a root for the this pointer in every member function. This is required so that, when a collection occurs, this pointers on the stack will be updated. An example of this is shown in figure 6. The macro SAVE_THIS constructs a root that references the this pointer. Unfortunately, current implementations of *cfront* prohibit taking the address of this. Therefore, the implementation of this macro is

```
class dnode : public node {
    node * center;
    ...
    virtual void copycollect(node * &);
    virtual void copycollect(dnode * &);
};

void dnode::copycollect(node * & n)
{
    ...
    n = new dnode(*this);  // safe, implicit conversion
    ...
}

void dnode::copycollect(dnode * & n)
{
    ...
    n = new dnode(*this);  // No conversion needed
    ...
}
```

Figure 3: Overloaded copycollect functions for a derived `node` type.

compiler dependent. Member functions that *cannot* directly or indirectly cause collections do not require this, however, omitting it is not recommended.

4 Efficiency

4.1 Overview

This dynamic memory management organization does not impact code that does not use it, thereby satisfying the first important efficiency criterion. The other is that code that does use the collector is also efficient.

The tests reported here were compiled with optimization enabled on a two processor VAXSTATION 3520, cfront 2.0 and the ULTRIX 3.0 C compiler. For comparison purposes many tests were also run on a Sun SPARCSTATION 1 running SUNOS 4.0.3 and cfront 2.0 and are so indicated.

In analyzing the performance of individual components we examine the following operations:

- allocating an object
- creating and destroying roots, both stacked and doubly-linked

```
/* Supplied in gc.h */
#define COLLECTION_MEMBERS(T)                              \
private:                                                   \
    static mallor freestore;                               \
    static void   gc();                                    \
    struct forward_addr {                                  \
        T * forward;                                       \
        forward_addr() : forward(0) { }                    \
    } fa;                                                  \
    void    set_forward(void * p) { fa.forward = p; }      \
    void *  get_forward()  { return fa.forward; }          \
public:                                                    \
    void *  operator new(size_t n)                         \
                { return freestore.get(n); }               \
    void    operator delete(void * p)    { }
```

Figure 4: The necessary members of a collected type.

- collecting a data structure

4.2 The Allocator

The time measurements shown in the graph of figure 8 are obtained as follows. Three allocators are compared: the testing version, the faulting version, and the standard `operator new` memory allocator with *malloc*. Each allocator was used to allocate 20 byte objects such as that shown in figure 7. After allocation, one byte was initialized in each object. This seems fair because uninitialized dynamically allocated objects should be rare. The test repeatedly allocates new objects until a fixed amount of memory has been obtained. The test was run to obtain 512k through 4M bytes. The custom allocators are parameterized by their internal chunk size. Chunks of 16k, 256k and 1M bytes were tested. Timing information was obtained with the `wait3` system call. The times reported are user time plus system time. The vertical bars show 95% confidence intervals.

The data presented in the graph of figure 8 show little difference between the two versions of the custom allocator. Both are roughly 2-3 times faster than the standard allocator. The faulting allocator with 16k chunks is slower than the other configurations due to the overhead of handling frequent write-faults.

4.2.1 Space Efficiency

In this context, space overhead is memory obtained from the operating system but not made available to the application. Examination of the process' *break*, before and after all the allocations, indicates how much space each allocator wastes. Under UNIX[1] the *break* is the

[1]UNIX is a trademark of Bell Laboratories.

```
/* Coded by the application programmer in node.h */
class node {
    ... // whatever required by the application
    ... // normal constructors, no destructor,
    COLLECTION_MEMBERS(node)
    virtual void copycollect(node * &);
};

/* Invoke parameterized macro to define the root class */
ROOT(node)

/* Coded by the application programmer in node.c */
DECLARE_GC(node)         /* define static data required for GC */
```

Figure 5: Defining a garbage collected class.

```
void node::membfunc()
{
    SAVE_THIS(node)
    ...
}
```

Figure 6: Defining a garbage collected class member function.

program's dynamic storage space limit. It grows toward higher addresses as the program obtains memory from the operating system. The `sbrk` system call with an argument of zero returns the current *break* without changing it. The total memory obtained from the operating system by each allocator is shown in figure 9.

In these tests the standard allocator was roughly 50% space efficient. In general, it's overhead depends on the request size. When requests are equal to or slightly larger than a power of two, the allocator suffers from severe fragmentation, obtaining $2n$ bytes from the operating system to provide n bytes to the application. When objects are slightly smaller than a power of two its overhead is very small.

4.3 Roots

4.3.1 Creating and Destroying a Root

Creating a root means allocating and constructing a pointer variable. Global and `static` variables are created once at the beginning of the program. The efficiency of constructing them contributes little to the efficiency of a program. The critical roots are those allocated on the stack including: local variables, by-value function parameters, function return values,

```
struct node {  /* sizeof node == 20 */
    static mallor * heap;
    void * operator new(size_t size) { return heap->get(size); }
    void operator delete(void * p)   { }
    char data[20];
};
```

Figure 7: The object used in allocator tests.

and temporaries.

Creating a pointer on the stack normally requires two instructions: decrementing the stack pointer and initializing the root. The stack pointer allocation is performed with one subtraction for all the local variables to the function. Destroying such a variable normally requires no instructions because the stack pointer is restored during the normal function return sequence.

Replacing the pointer with a `root` adds three instructions: two for maintaining the stack and one to initialize the root to NULL. Unlike a normal pointer variable, a `root`'s pointer component must not be left uninitialized. The root destructor requires one instruction. Finally, the compilers use for these tests move an unnecessary value into `r0` for the expression value. Creating, initializing and destroying a root, which should take four instructions, takes 5 under these compilers. The figures in table 1 show the performance of root allocation and destruction.

Table 1: Efficiency of creating `roots` compared to simple pointers.

Operation	Time
Create/initialize/destroy 2,000,000 pointers	11.8s
Create/initialize/destroy 2,000,000 roots	14.2s
Create/initialize/destroy 2,000,000 droots	23.1s
Startup, termination and function call overhead	11.0s
Time per pointer	0.4μs
Time per root	1.6μs
Time per droot	6.1μs

As predicted, creating roots is on the order of four times more expensive than creating pointers. It requires three machine instructions per root that are not required by a simple pointer.

Multiple roots will be created and destroyed when there are multiple local variables of type root. An unsophisticated compiler might not optimize their construction and destruction. However, the compiler could safely eliminate all but one `mov` to set the list head, and all but one `mov` to restore the old list head. Thus, the obvious code to construct and destroy

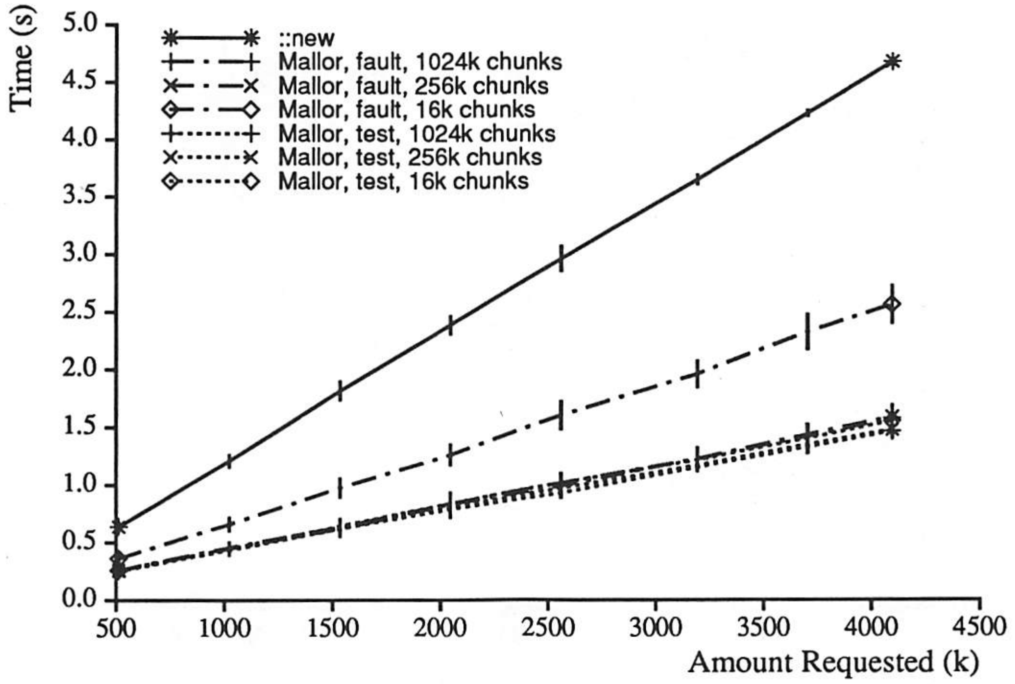

Figure 8: Time to Allocate and Touch: 20 Byte Requests, SPARCSTATION 1

Mallor: means the custom memory allocator
fault: the version that uses write-faulting to avoid the test
test: the version with an explicit bound check

n roots will use $3n$ instructions, however, optimization reduces that to $n + 2$ instructions.

4.4 Collecting

A garbage collection pass visits every root, and then copies every node reachable from the root. A garbage collection includes processing for the *flip* plus the deep copy of the data structure. The flip is constant time and fast. Deleting from-space is very fast (tens of instructions, not hundreds.) Deeply copying the data structure requires a virtual function call per pointer and an allocation/copy per object. It is difficult to estimate the amount of time this will take because the application defines the copy constructor.

Calls to the allocator take 6 instructions, except when a fault occurs. With 256k chunks faults will be very rare, there would be four if a megabyte were copied. Twenty byte objects, such as those benchmarked in this paper, would require 5 instructions each to copy. If many objects are copied this expense will be significant. The key in copying collection is to wait to collect until many objects have died. That way copying is very fast.

For comparison purposes the collector was benchmarked collecting a graph of 20 byte nodes on both the VAXSTATION and a SPARCSTATION. The results are shown in table 2.

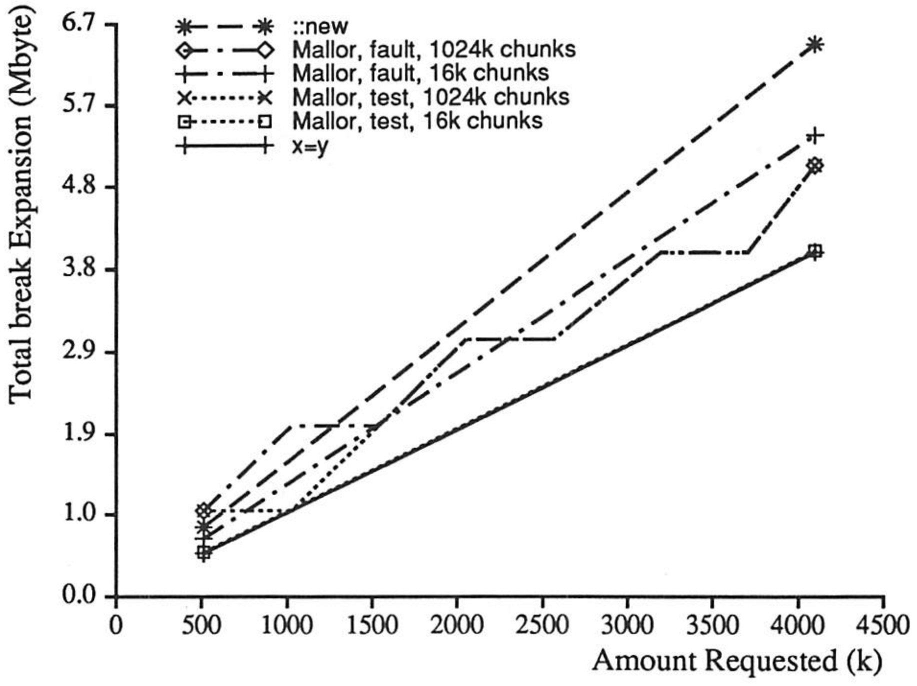

Figure 9: Total memory allocated, SPARCSTATION 1.

5 Conclusion

5.1 Advantages

Reclaiming dynamically allocated objects can be difficult. With the collector, inaccessible memory is recycled and live objects are compacted. This increases programmer productivity by removing from the application programmer responsibility for deallocating data. Compaction may reduce paging and improve virtual memory performance.

Many copying collector allocators fix the size of the free-store. This allocator will more closely track the actual amount of memory required since, after a collection, it is off by no more than one chunk. The behavior of other allocators is trivially emulated by using a very large chunk size and disallowing expansion.

This collector demonstrates a new way of tracking roots of the data structure. Allocating and initializing n roots in a stack frame requires approximately $n+2$ more memory references than simple pointer allocation and initialization. These roots must be initialized since following uninitialized roots would lead to errors. Iterative code, such as list traversal, when no roots are constructed within the loop, suffers no performance penalty under this scheme.

The allocator supplied with this package is fast. The measurements of §4.2 show that the allocator supplied with this system is much faster than the allocator provided in the standard libraries. This is unsurprising since copying collection permits very efficient allocation.

Table 2: Collector Efficiency Measurements

Nodes have 12 bytes of data, 4 bytes of forwarding pointer, and 4 bytes of *vptr*.
The data structure of $2^{17} - 1$ nodes requires 2559k bytes.

Operation	Time to Complete	
	VAXSTATION 3520	SPARCSTATION 1
Collect "null", 1 root, 1 chunk, no data	$58.5\mu s$	$11.5\mu s$
Build a binary tree: $2^{17} - 1$ nodes	$4.2s$	$2.4s$
Build and collect the tree	$12.3s$	$4.2s$
Time to copy $2^{17} - 1$ binary nodes	$6.0s$	$1.8s$
Time per node (minus overhead)	$45.8\mu s$	$13.7\mu s$
Nodes copied per second	21,845	72,992
Kbytes copied per second	426	1425

5.2 Limitations

One problem, not with this implementation but with copying collection for C++, is that objects are never individually deallocated. Without deallocation of individual objects destructors cannot be called for collected objects. A programmer who provides a collected `class` with a destructor is likely to be surprised when the destructor is never invoked. Never examining dead objects increases the efficiency of copying collectors. This explains why, during a collection, copying collectors do work proportional to the number of living objects rather than number of dead plus living objects. If copying collection is preferred, this cannot be seen as a disadvantage. It is only a disadvantage because it is a divergence from the C++ style. As Koenig observed, destructors must be forbidden for copy-collected objects [Koe90].

The system does not support arrays of collected objects, though it supports arrays of pointers to collected objects. As a policy decision, all objects of a collected type must be dynamically allocated. This allows the garbage collector to avoid checking to see if each object is dynamically allocated. A global or `static` object can be simulated using a global `root` that always references the same object. Non-dynamic objects can be supported at the cost of reduced collection efficiency.

The collector has a scheme for working with foreign roots. However, since destructors are not called for collected objects, a foreign (doubly-linked) root must not be inside an object that is reclaimed with copying garbage collection. If a collected object contained a `droot` to another object, the root would continue to exist even after the containing object was deallocated. Objects containing foreign roots must be reclaimed another way, such as manually.

The prototype system can not prevent the programmer from creating *dumb* pointers. Dumb pointers are those that do not track themselves using the lists. Currently, C++ does not enable an application programmer to prevent the creation of dumb pointers to a class.[2] In a compiler-based implementation this will not be an issue.

5.3 Summary

Efficient management of dynamic data is difficult. Only in simple cases can programmer-controlled deallocation be safe, efficient and simple. Reclamation of generalized dynamic graph data structures requires edge traversal to identify unused blocks. Unless blocks are reused, objects are scattered in memory causing excessive paging and using an unnecessarily large amount of backing store.

There are two predominant classes of reclamation algorithms: mark-and-sweep and copying. Both must normally be able to recognize pointers to collected objects. Mark-and-sweep collectors take one pass to mark all living objects and one pass to deallocate inaccessible ones. Another pass may be used to compact objects. Conservative collectors can be implemented without compiler support. These collectors need not differentiate between pointers and integers. They reclaim most, though not all, of the inaccessible memory.

This paper has presented a copying collector and its C++ implementation. The collector will work with lists, trees, DAGs, or cyclic graphs. It incorporates very fast allocation and a novel way of tracking roots that does not require tagged pointers or integers. The collector's work is proportional to the amount of living data, therefore when most objects die, it is highly efficient. This is the only copying collector for C++ known to the author that requires neither tags nor an object table, and that supports polymorphic type hierarchies.

This implementation of the collector requires no modifications to the compiler, but it does require assistance from the programmer. Another version in the compiler would require changes to the language. Such a collector could be more efficient than this one.

The system is composed of encapsulated data structures making it appropriate for an object-oriented imperative programming language. This collector shows a way that garbage collection can be made an efficient, non-invasive part of the C++ programming language. Under this scheme the efficiency of reclaiming a data structure depends exclusively on the complexity of the data structure.

In this system we have accomplished the following:

1. added efficient and reasonably convenient automatic storage reclamation to C++,
2. found an organization for garbage collection in C++ that remains within the philosophy of the language, and,
3. developed a platform for research into new techniques and algorithms, particularly in copying collection and virtual memory issues.

Availability

The prototype collector has not yet been made publicly available. When we have a more robust and easy-to-use implementation, it will be made available via anonymous ftp.

[2]Guilmette has proposed C++ language changes that would accomplish this [Gui91], but it is unclear whether or not the C++ standardization committee X3J16 will adopt the proposal.

A longer report that describes this work can be obtained via anonymous ftp from midgard.ucsc.edu (128.114.14.6). It is in pub/tr/ucsc-crl-90-19.ps.Z. This includes a printed copy of the source code that implements the prototype. Printed copies of the technical report are available. Write to:

Jean McKnight
Technical Librarian
Baskin Center for Computer Engineering & Information Sciences
University of California
Santa Cruz, CA 95064

Internet: jean@cis.ucsc.edu

References

[Ame89] ANSI C Standard, 1989. American National Standard X3.159-1989.

[App87] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, June 1987.

[Bak78] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[Bar88] Joel F. Bartlett. Compacting garbage collection with ambiguous roots. Technical Report 88/2, Digital Equipment Corporation, Western Research Laboratory, Palo Alto, California, February 1988.

[Bar89] Joel F. Bartlett. Mostly copying garbage collection picks up generations and C++. Technical Report TN–12, DEC WRL, October 1989.

[BW88] Hans-Juergen Boehm and Mark Weiser. Garbage collection in an uncooperative environment. *Software—Practice and Experience*, 18(9):807–820, September 1988.

[CDG+88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report. Technical report, Digital Systems Research Center and Olivetti Research Center, Palo Alto, CA, 1988.

[Ede90] Daniel Edelson. Dynamic storage reclamation in C++. Technical Report UCSC-CRL-90-19, University of California at Santa Cruz, June 1990. M.S. Thesis.

[EP90] Daniel Edelson and Ira Pohl. The case for garbage collection in C++, August 1990. Workshop on Garbage Collection in Object-Oriented Programming Languages, in conjunction with OOPSLA/ECOOP '90.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley Publishing Company, February 1990.

[FY69] R. Fenichel and J. Yochelson. A LISP garbage-collector for virtual-memory systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley Publishing Company, Reading, MA, 1983.

[Gui91] Ron Guilmette, February 1991. Usenet comp.lang.c++ article: Re: Smart pointers and stupid people.

[HD90] Richard Hudson and Amer Diwan. A copying collector for Modula-3 and Smalltalk, 1990. private communication.

[Knu73] Donald E. Knuth. *The Art of Computer Programming*, volume 1. Addison, Wesley, Reading, Mass., 1973. Second ed.

[Koe90] Andrew Koenig. Objects reclaimed by a copying collector must not have destructors, 1990. private communication.

[LH83] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1988.

[Ste84] Guy L. Jr. Steele. *Common Lisp: The Language.* Digital Press, Burlington, MA, 1984.

[Str87] Bjarne Stroustrup. The evolution of C++ 1985 to 1987. In *Usenix C++ Workshop Proceedings*, pages 1–22, Santa Fe, NM, November 1987. Usenix Association.

[Ung84] David Ungar. Generation Scavenging: A non–disruptive high performance storage reclamation algorithm. In *ACM SIGPLAN/SIGSOFT Symposium on Practical Software Development Environments*, pages 157–167, Pittsburgh, PA, April 1984. Association for Computing Machinery.

[Ung86] David Michael Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* The MIT Press, Cambridge, MA, 1986.

Type Identification in C++

Dmitry Lenkov
Michey Mehta
Shankar Unni

California Language Laboratory,
Hewlett-Packard Company,
19447 Pruneridge Avenue,
Cupertino, CA 95014.

E-Mail: {dmitry|mnm|shankar}@cup.hp.com

ABSTRACT

Many applications and class libraries require a mechanism for run-time type identification and access to type information. This paper describes a general type identification mechanism consisting of language extensions and library support. We introduce the following language extensions to support type identification uniformly for all types: a new built-in type called **typeid**, and three operators **stype** (static type), **dtype** (dynamic type), and **subtype** (subtype inquiry). We also describe a library class called **TypeInfo**, which is used to access compiler generated type information. Special member functions of the **TypeInfo** class are used to extend the compiler generated type information. An implementation strategy is presented to demonstrate that the proposed extensions can be implemented efficiently. We compare our proposal with previous work on runtime type identification mechanisms.

1. Introduction

There have been various attempts made in C++ to implement a method of type identification for objects and a mechanism to access additional type information[3][4][5]. There are several reasons why such identification is needed.

- Support for accessing derived class functionality
  Many of the commonly available C++ class libraries (such as NIH[4], InterViews[6], and ET++[5]) consist of an inheritance hierarchy with a root class (such as the Object class in NIH). When dealing with pointers to this root class, a common operation in these toolkits is to determine if a pointer points to an object of a derived class. If so, the pointer is castdown to the derived class so that a derived class member function may be invoked. Since C++ performs its type checking at compile time, type information is not available at runtime, and each toolkit uses different mechanisms for determining the actual type of the object being dereferenced. When the root class is a virtual base class (as in NIH), since the castdown is not permitted by C++, the library must invent mechanisms to circumvent this restriction. We show that our type identification scheme supports subtype queries and castdowns.

- Support for Exception Handling.
  The exception handling mechanism[1][2] requires type identification at run time, in order to match the thrown object with the correct **catch** clause. The exception handling mechanism is

an example of an implicit use of the type identification mechanism. Since a **catch** clause can catch a type which is a base class of the thrown object, it is necessary for the compiler to generate information about inheritance hierarchies for our proposed **subtype** operator to work.

- Support for Accessing Type Information.
  There are various class-specific actions that are difficult to achieve using the normal virtual function mechanism. For example, consider the following task: count (or do some similar task) for all nodes of a particular type in a tree of polymorphic objects.

- Support for Libraries and Toolkits.
  Once the type of an object has been determined at runtime, it may often be necessary to get further information about the type. For example, as described in [3], applications may need to know the names of classes and their inheritance hierarchy, if a customization mechanism uses class and instance names. Our proposal describes library support for getting additional information about a type.

In this paper, we examine the problem of type identification in C++. We propose language extensions that will support type identification, and describe methods of implementing our proposal. The goal of our scheme is to create a uniform mechanism for the creation of and access to type information.

2. Language Extensions for Type Identification

In this section we describe several extensions to C++ necessary to support functionality required by the applications mentioned in the introduction.

2.1. The subtype Operator

Applications often require the ability to determine dynamically if a pointer points to an object which is a subtype of a given type and, in certain cases, cast the pointer down to the given type. The **subtype** operator lets the programmer examine the inheritance relationship of object types at runtime. For example,

```
subtype(A, p)
```

determines if the actual type of the object pointed to by "p" is a subtype of type A.

The **subtype** operator is a predefined operator that takes a type name as the first parameter and a pointer as the second parameter. It returns a result of type int. The result is 1 if the dynamic type of the object being pointed to is a subtype of the type provided as the first parameter (otherwise 0 is returned). Note that a type is a subtype of itself. Consider three classes:

```
class List {...};

class SortedList: public List {
   ...
   Key least_key();
}

class LenSortedList: public SortedList {
   ...
   int length();
}
```

Here are some examples of how the **subtype** operator can be used.

Example 1:
```
List* l_p = // initialize
...
l_p = // point to some other list
...
if( subtype( SortedList, l_p)) {
   Key k = (SortedList*) l_p -> least_key();
   ...
}
...
if( subtype( LenSortedList, l_p))
   cout << ((LenSortedList*) l_p -> length());
```

Another example is calling a function that requires an actual parameter which is a derived class.

Example 2:
```
void func( LenSortedList *);
...
if( subtype( LenSortedList, l_p))
   func( (LenSortedList*) l_p);
```

In the previous two examples the castdown operation was used to allow functionality defined on subtypes to be used. However, the subtype operator also has applications that do not require a cast-down operation. Consider:

Example 3:
```
void sort( List*);
...
List *l_p = // initialize
...
if( !subtype( SortedList, l_p))
   sort( l_p);
```

Consider another example:

Example 4:
```
void other_func( OtherType *);
...
OtherType p = // initialize
...
if( subtype( SortedList, l_p))
   other_func( p);
```

In this example, the functionality associated with the SortedList subtype is invoked as in example 2. However actual actions take parameters of types other than SortedList. Thus the castdown operation is not needed.

C++ types fall under three different categories with regard to the subtype operator: polymorphic classes (those that have virtual functions), simple types, and non-polymorphic classes. For polymorphic classes the behavior of the **subtype** operator is illustrated above. A simple type (int, int (*)(), etc.) has no subtype (other than itself). Thus the **subtype** operator establishes equality for them with the result defined statically at compile time. For example,

Example 5:

```
typedef int* int_p;
...
int_p ptr = // initialize
...
if( subtype( int, ptr))
   //action
```

The use of the **subtype** operator for non-polymorphic classes is limited because only statically defined types can participate in the operation. If the three classes defined above are non-polymorphic (no virtual functions are declared) then in examples 1 through 4 the result of **subtype** will be 0 (and defined at compile time). Thus the **subtype** operator is not useful for simple types and non-polymorphic classes. It may be desirable to produce a warning if a **subtype** operation results in a compile time value, since the programmer may not be aware that the class is non-polymorphic.

2.2. Castdowns

In the introduction we noted that the ability to safely access type related functionality is an important requirement for application and library developers. The first two examples above show that such access in many cases requires that a pointer be cast down to a derived type. In those examples, it can be safely done at compile time. However, if virtual base classes are involved then it cannot be done statically.

Currently the C++ language does not allow a pointer to a base class to be cast down to a derived class pointer if the base class is virtual, or if there is a virtual derivation between the base class and the derived class. The reason for this is that it would require an implementation to maintain pointers from virtual base classes to derived classes. The introduction of the **subtype** operator requires us to keep information about subtype relationships, and the information for pointer conversions can be stored in these data structures. Therefore, we propose that this casting restriction be removed. Note that we cannot remove this restriction for non-polymorphic classes; however, for such classes the **subtype** operation would always fail anyway (since the static type would be used). Since cases of static casting and dynamic casting can be distinguished semantically and do not require a syntactic distinction, the introduction of an additional operator specifically for dynamic casting is not necessary.

Let us modify the classes from the previous section:

```
class List {...};

class SortedList: virtual List {
   ...
   Key least_key();
};

class LenList: virtual List {
   ...
   int length();
};

class LenSortedList: SortedList, LenList {...};
```

Now both casts in example 1, `(SortedList*) l_p` and `(LenSortedList*) l_p`, become illegal in the current definition of C++. We propose to extend the definition of casts and make these casts legal.

The proposed extension of the cast definition raises an additional issue. What happens if "(LenSortedList*) l_p" is used without doing "subtype(LenSortedList, l_p)", and it turns out that l_p points to an object of a class which is not a subtype of LenSortedList? Currently if one attempts:

```
B* b_p = //initialize
C* c_p = (C*) b_p;
```

where B and C are unrelated but have a common parent, an unchanged value of b_p is assigned to c_p. It is reasonable to do the same in the case of dynamic casting.

2.3. The Type Identification Scheme

Some of the applications described in the introduction would require a unique identifier to be associated with a type. The primary component of this type identification scheme is the predefined type called **typeid.**

2.3.1. The typeid Type

The **typeid** type is a simple predefined type, similar to **int** or **void***, with a few operations defined on it. Expressions evaluating to the **typeid** type can be compared for equality and inequality. Variables of the **typeid** type can be assigned or initialized with an expression of the **typeid** type. No other operations are allowed. Each unique type in an application has a unique value of the **typeid** type associated with it. We define two operators which return values of type **typeid.**

stype returns the type identifier (**typeid** value) for the static type of an expression. It can also be applied to a type name and returns the type's **typeid** value. The **dtype** operator can be applied to any expression that evaluates to a pointer to a type. If the pointer points to a polymorphic class, **dtype** returns the type identifier (**typeid** value) of the actual type of an object pointed to by this pointer. Note that this type must be determined dynamically. If the pointer does not point to a polymorphic class, **dtype** returns the **typeid** value of the static type pointed to by the pointer definition.

Example 6:

```
List* l_p = new SortedList;
int num_Sorted_Lists = 0;
...
typeid t  = dtype(l_p);
if (t == stype(SortedList)) num_Sorted_Lists++;
```

The reason that **stype** and **dtype** are not predefined member functions is the same reason that **sizeof** is not a member function: both identify a fundamental property of types, as opposed to an operation on objects of those types. On the other hand, both can be applied to any types including types such as (int* (*) ()).

An alternative to the **stype** operator is to allow an explicit conversion of any type to **typeid.** However this would also require the conversion of type names to **typeid.** The above example would look like:

```
List* l_p = new SortedList;
int num_Sorted_Lists;
...
typeid t  = dtype(l_p);
if (t == typeid(SortedList)) num_Sorted_Lists++;
```

2.3.2. Accessing Additional Type Information

Given a **typeid**, programmers may wish to get information about the underlying type; programmers may also wish to extend the type information automatically generated by the compiler (for example, they may wish to store the name of type).

We propose a standard library function called **get_typeinfo** to convert a **typeid** into a pointer to the **TypeInfo** object. The **TypeInfo** class contains various member functions to get information about the underlying type (if it is a class). The specification of the **TypeInfo** class is shown in section 3.

Note that the **typeid** type is really the same as a **TypeInfo***, and **TypeInfo*** could be used in its place for the extensions described above. The advantage we gain from separating these two types is that we make a clear distinction between the types recognized by the language and the types recognized by the standard library. In addition, the use of **TypeInfo*** for type identification is unsafe because a variable of this type can be assigned values unrelated to actual type identifiers.

3. Library Support for Type Identification

When the user calls the type inquiry function get_typeinfo(), the result is a pointer to a **TypeInfo** object. In this section we describe this class and the type inquiry function get_typeinfo(). We also describe how a class user can extend the amount of information available about this class.

3.1. The TypeInfo Class

The implementation of the language features described in the previous section will require an implementation to store some information about each class. This information can also be accessed using the **TypeInfo** class interface described below. We expect that the C++ library standardization effort will determine the minimum functionality to be provided by all implementations.

```
class TypeInfo {
public:
   int sizeof();                            //size of type
   int get_num_base_classes();              //Number of base classes
   typeid get_base_class( int pos);         //typeid of specified base class
   int is_virtual_base_class( int pos);     //is specified base virtual?
   int visibility_of_base_class( int pos); //public(==2), protected(==1)
                                            //or private(==0) base class?

   //The routines are used to extend the compiler generated type information
   AuxTypeInfo* get_aux_typeinfo( typeid key);
   int add_aux_typeinfo( AuxTypeInfo *info, typeid key);
private:
   //Actual implementation
};
```

The type inquiry function is specified as follows:

```
TypeInfo* get_typeinfo(typeid)
```

3.2. Extensibility

Clearly, there needs to be a way of to define and access more than just the minimal type information provided by **TypeInfo**. For example, it is possible that a developer may wish to determine the name of a class at runtime. The information stored and the association of that information with the class **TypeInfo** object needs to be examined in detail. This section describes mechanisms whereby a user may extend the type information associated with a class. We believe that it is best to allow the class library creators and users to specify what information needs to be associated with each type.

The following mechanism is used to extend the type information associated with a type.

- We provide a member function called "add_aux_typeinfo" in the **TypeInfo** class. This member function is used to attach additional type information to the minimal type information generated for a type.
- We provide a member function called "get_aux_typeinfo" in the **TypeInfo** class. This member function is used to retrieve any additional type information that a user may have attached to a type.
- It is reasonable to expect that multiple users may wish to attach auxiliary type information to the same type. Therefore, the notion of a "key" is required. A "key" is used to distinguish between multiple auxiliary type information objects attached to the same type.

Consider an example:

```
// User wants to add a "name" field to the TypeInfo for class Widget

//See section 3.3 for an explanation of the AuxTypeInfo class
class NameInfo : AuxTypeInfo {
   char *name;
public:
   NameInfo(char* n): name(n){};
};

NameInfo NameInfoObject = "Widget";
// Attach additional type information for "Widget"
get_typeinfo( stype(Widget)) -> add_aux_typeinfo( &NameInfoObject, stype(NameInfo));

// Assuming the user has installed name information in Widget, and
// all classes derived from it, here is how a user could dynamically
// find out the name of a class.
Widget* w = // initialized to something;
char* name = (NameInfo*) (get_typeinfo( dtype(w)) ->
                  get_aux_typeinfo( stype(NameInfo))) -> name;
```

The extensibility scheme we have proposed is essentially a convenient method of adding a static member (in fact, a virtual static member) to an existing type, without having to modify the type in any way. Individual users can certainly come up with various methods of accomplishing the same result, but the goal here is to propose a *uniform* method for extending type information.

3.3. The AuxTypeInfo Class

Any additional type information should be defined as a class derived from AuxTypeInfo. Instances of this are used to link the auxiliary type information objects. See section 4.5 on additional information about the implementation of extensibility. The AuxTypeInfo class is defined as follows:

```
class AuxTypeInfo {
    // next auxiliary type info
    AuxTypeInfo* next;
    // The type of the class derived from this class
    typeid key;
};
```

3.4. Useful Macros

The library header file can define macros so that the additional "key" parameter can be automatically generated. For example:

```
#define ADD_TYPE_INFO( TYPENAME, INFONAME, INFO_PTR) \
 get_typeinfo(stype(TYPENAME))->add_aux_typeinfo(INFOPTR,stype(INFONAME));

#define GET_TYPE_INFO( INFONAME, OBJECT_PTR) \
 (INFONAME*) (get_typeinfo(dtype(OBJECT_PTR))->get_aux_typeinfo(stype(INFONAME)))
```

These macros can be used to rewrite the example described in section 3.2.

```
class NameInfo : AuxTypeInfo {
    char *name;
public:
    NameInfo(char* n): name(n){};
};

NameInfo NameInfoObject = "Widget";
// Attach additional type information for "Widget"
ADD_TYPE_INFO( Widget, NameInfo, &NameInfoObject)

// find out the name of a class.
Widget* w = // initialized to something;
char* name = GET_TYPE_INFO( NameInfo, w) -> name;
```

4. Implementation Strategy

In this section we describe implementation schemes where the compiler will automatically create the type identification information necessary to support the type inquiry operators. We also describe the implementation we use to allow programmers to extend the compiler generated type information with additional type information. Our goal in this section is to demonstrate that reasonable implementation schemes exist. However we assume that actual implementations will optimize these schemes for performance. Although the implementations schemes proposed in this section are geared towards AT&T C++ front end based compilers and translators, we consider them portable to other C++ implementations.

In analyzing various possible implementation schemes, we kept the following goals in mind:

- The implementation scheme should be portable to a variety of C++ implementations.
- There should be no space or execution penalty for users who do *not* use type inquiry operators.

- Reasonable space and execution performance should be expected when using type inquiry operators.
- When a program uses type inquiry operators the execution cost should be paid only when (and if) these are actually used at runtime. Any startup cost should be minimized.
- We wanted a scheme that would work with both "munch" and "patch" (see next section).

4.1. Terminology

The implementation section of this paper uses terminology that may not be familiar to everyone.

- *Munch* and *Patch* : Munch and Patch are schemes used by AT&T C++ front end based implementations to ensure that all static objects are appropriately initialized before the main program begins. After a program is linked, a "munch" implementation scans the resulting executable for special symbols and constructs additional data structures which are then relinked into the program. In a "patch" implementation, the executable resulting from a link is also scanned for these special symbols. But instead of constructing additional data structures, "patch" fixes existing data structures, for example, linking some of them together.
- *vtables*: "vtables" are tables, or data structures, which support virtual function calls. A polymorphic object will contain one or more pointers to one or more such tables.

4.2. Implementation Details

We now provide some details of a possible implementation scheme. In section 2 we described the built-in **typeid** type and in section 3 we described a library routine **get_typeinfo** which will convert a **typeid** into a **TypeInfo***. A **typeid** is really equivalent to a **TypeInfo***, and in the rest of this section we will always use the **TypeInfo** class name.

Our overall implementation strategy is:

- One **TypeInfo** object per type:
  The type inquiry operators return a pointer to a *unique* **TypeInfo** object associated with the type. The reason we need to guarantee one unique object is so that pointer comparisons can be used to determine whether two types are the same.
- **TypeInfo** objects are only allocated if necessary:
  Our implementation scheme attempts to minimize the number of **TypeInfo** objects which are allocated, since we do not need to allocate one for every single type. Allocating a **TypeInfo** object for every single type we encounter in a program is not necessary, since a compiler can determine whether or not the **TypeInfo** object for a type is accessible at runtime.

4.2.1. Allocation of TypeInfo objects

TypeInfo objects can be referenced at runtime for any of the following reasons:

1. We must have **TypeInfo** objects for the static types of any types used in type inquiry operators. For types which are classes we must also allocate **TypeInfo** objects for each ancestor in the class hierarchy. This is needed to allow traversal of the ancestor hierarchy of a class in order to support **subtype** inquiries and the **TypeInfo** class functionality. This also supports the exception handling mechanism.
2. We must have **TypeInfo** objects for all derived classes of polymorphic base classes on which the user performs a dynamic type inquiry operation (i.e **dtype** or **subtype**). Since a derived class can be defined in a compilation unit which is not visible to the compilation unit containing a type inquiry operator, **TypeInfo** objects have to be emitted for all polymorphic classes.

Let us now examine what rules a compiler must follow when deciding whether or not to emit type information for a given type. There is a unique **TypeInfo** object per type, and this object will contain additional information about class types. In a "munch" implementation, the mechanism for guaranteeing *unique* **TypeInfo** objects per type relies on using tentative definitions (available in both K&R C and ANSI C). This implies that the **TypeInfo** object can be initialized exactly once in one of the object files submitted to the linker, or not initialized at all (in which case the object will get a default initialization). In a "patch" implementation we may sometimes emit multiple **TypeInfo** objects for the same class, but at runtime we will always reference the same **TypeInfo** object.

4.2.2. Polymorphic Classes

Each polymorphic object contains a pointer to a vtable. For a polymorphic class, cfront emits one vtable for the class in the compilation unit which contains the definition of the first non-inline non-pure function; if there is no such function, then a vtable is emitted in every compilation unit in which one is required.

We will implement dynamic typing by storing a pointer in the vtable to the appropriate **TypeInfo** object. Note that every single vtable must contain this pointer, whether or not we see a **dtype** in the current compilation unit (since we do not know if a **dtype** was done in another compilation unit).

In cases where a unique vtable can be emitted for a class, we will emit a definition for the corresponding **TypeInfo** object in the same compilation unit. We expect that most classes will fall into this category.

In cases where cfront emits multiple vtables, we will have to allocate a tentative definition for the **TypeInfo** object in a munch implementation and an initialized definition for the **TypeInfo** object in a patch implementation in each such compilation unit.

Note that the use of vtables to store pointers to **TypeInfo** objects does not allow the optimization which occasionally allows derived classes to "share" vtables with their parents.

4.2.3. Other Types

This section describes how we handle all other types.

- Non-polymorphic classes:
  Whenever a non-polymorphic class is used in a type inquiry operation we must emit a tentative **TypeInfo** object for the class in a munch implementation, and an initialized definition for the **TypeInfo** object in a patch implementation. We must also do the same for each class in the ancestor hierarchy. See the following sections for more information on initialization of **TypeInfo** objects in munch and patch implementations.
- Non-class types:
  This includes built-in types (e.g. int), arrays, pointers, and references. Whenever such a type is referenced in a type inquiry operation we allocate a tentative definition for the corresponding **TypeInfo** object. Note that such types have no additional information associated with them, and the default initialization of these objects to 0 is acceptable. The only use which can be made of these **TypeInfo** objects is address comparison, and the installation of auxiliary type information.

4.2.4. Patch Implementation

We have described some cases where we need to emit multiple initialized **TypeInfo** objects. This section describes how we always manage to return a pointer to the same **TypeInfo** object (figure 1).

- Assume that each **TypeInfo** has a field within it called "RealTypeInfo" of type **TypeInfo*** which is initialized to 0 at compile time for types that do not have unique TypeInfo objects. For a type which has a unique **TypeInfo** object, we will simply make its "RealTypeInfo" field point to itself at compile time.

- Consider a class X for which we cannot find a unique place to initialize the **TypeInfo** information. Assume we allocate and initialize **TypeInfo** objects in file1 and file2 called TypeInfo_X_file1 and TypeInfo_X_file2, respectively.
- For the expression **stype(X)**, the equivalent C code we generate is:

```
C++ code                C code
========                ======
stype(X)                TypeInfo_X_file1.RealTypeInfo (if in file1)
stype(X)                TypeInfo_X_file2.RealTypeInfo (if in file2)
```

- The "patch" tool will notice that there are two TypeInfo objects for X, arbitrarily pick one of them, and make the "RealTypeInfo" fields of both objects point to the chosen version. Note that since we have allocated initialized objects, patch is allowed to modify them in the object file itself.

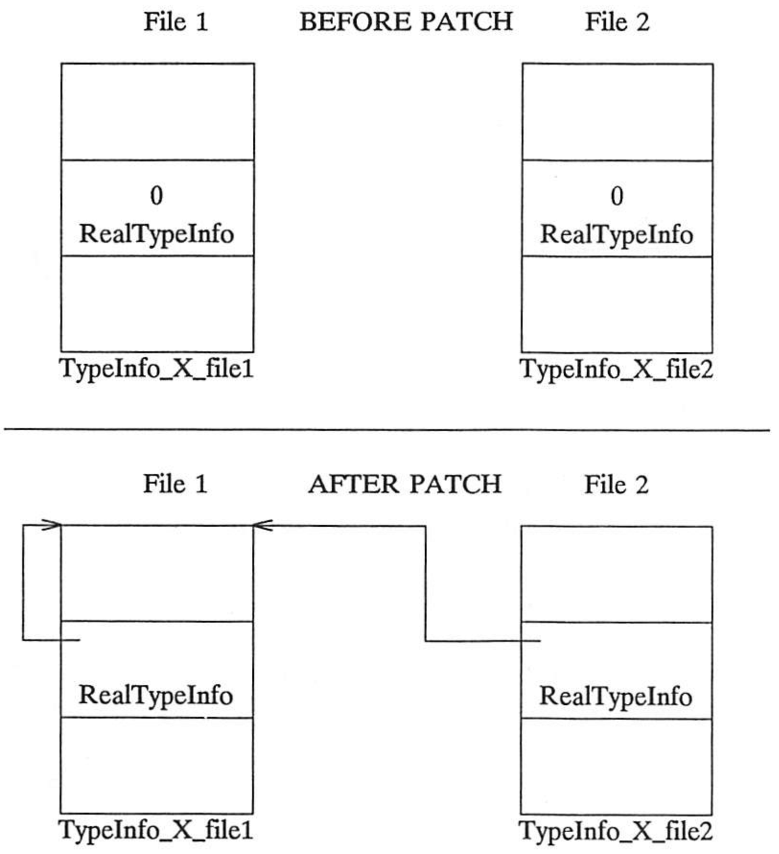

Figure 1: Patch Implementation

The primary advantage of this scheme is that there is no startup cost. Note that this same scheme cannot be used by munch without incurring some runtime cost, since there is no way for munch to initialize the "RealTypeInfo" fields without generating some code to execute at runtime. The next section describes a scheme for munch which involves no runtime initializations.

4.2.5. Munch Implementation

We have previously described that we will emit a tentative definition for the **TypeInfo** information for certain classes if we cannot find a unique compilation unit in which to perform the initialization. At "munch" time, how do we initialize these uninitialized objects? One approach is to emit enough information in the symbol name itself, so that "munch" can allocate an initialized definition in the object file it creates. For example, the symbol used for class X could be "TypeInfo_1X4base6window" if class X had base classes "base" and "window".

4.3. Implementation of Extensibility

Section 3 describes how the user can attach auxiliary type information to the information already stored for each type. This section provides some information on how to implement this feature.

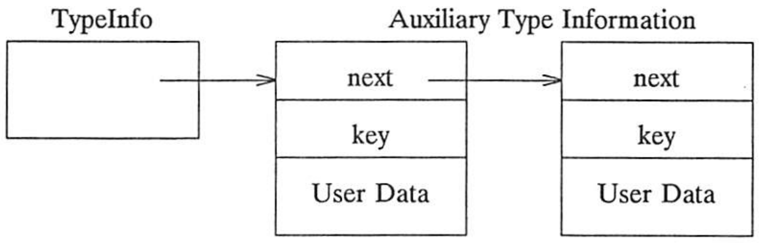

Figure 2: Attaching Auxiliary Information to **TypeInfo**

4.3.1. The Role of AuxTypeInfo

The example in section 3.2 shows the user inheriting NameInfo from **AuxTypeInfo**. The first parameter of the **add_aux_typeinfo** member must be an object which is derived from **AuxTypeInfo**. This section describes the reasoning behind this requirement.

As shown is Figure 2, the implementation of auxiliary type information is essentially a chain of objects. Each entry in the chain contains two pieces of information in addition to the object supplied by the user: a "next" field, and a "key" field. We did not want the add_aux_typeinfo routine to have to allocate memory for these additional fields. By requiring the user to inherit from **AuxTypeInfo** we guarantee that each auxiliary information object supplied by the user will contain these two fields. Note that the user may not add the same auxiliary object to two different lists, since we are chaining the actual object on the list, not a pointer to the object.

4.3.2. The "key" Parameter

Both the **add_aux_typeinfo** and **get_aux_typeinfo** member functions require a parameter called "key", which is of type **TypeInfo***. This section describes the rationale behind this parameter.

We would like to allow different components of a system to attach auxiliary type information for the same type. The "key" parameter is the mechanism for guaranteeing that the information added by a call of add_aux_typeinfo is the same information retrieved by a call of get_aux_typeinfo.

We considered various options for "keys" that the user may associate with the auxiliary type information object, including numbers and strings. Our proposal is to use the **typeid** of the auxiliary information structure as the key. Note that this means that two different users may not attach two auxiliary information objects of the same type to the same chain, since the keys would be the same.

4.4. Shared Library Considerations

Shared libraries are a mechanism for multiple programs to share the same copy of routines linked into a shared library; linking with shared libraries will generally result in much smaller executables files than linking with archive libraries. Shared libraries add complexity to the implementation of type identification. Although we do not go into any details of how shared libraries work (since vendors differ in their implementations), we make the following assumptions about shared libraries:

- At link time, a tool like "munch" or "patch" will not get a complete picture of all object files which belong in this executable, since shared libraries can be explicitly loaded at runtime (for example you may may load a set of graphics routines which depend on the output device you are using).
- When a shared library is created, we assume that there will be some mechanism which will allow us to run "munch" or "patch" on the shared library.
- We assume that the shared library mechanism allows us to specify a routine that will be executed when the library is first loaded.

The "patch" and "munch" schemes previously described rely on being able to process a "complete" executable; since a link involving shared libraries results in an "incomplete" executable we need to make modifications to our schemes.

The previous section describes two conditions under which we cannot initialize a unique **TypeInfo** object at compile time:

1. Polymorphic classes which do not have unique vtables
2. Non-polymorphic classes

We now describe how this would be handled in a shared library implementation. Both implementations suggested below will have some runtime initializations being performed.

4.4.1. Patch Implementation for Shared Libraries

The scheme of having a "RealTypeInfo" field does not work for shared libraries, because when a shared library is loaded it would be difficult (and expensive) to make each such field point to the appropriate **TypeInfo** object. The shared library patch algorithm looks like this:

1. At compile time we emit a tentative definition for a **TypeInfo** object which cannot be initialized in a unique file. We also emit an initialized definition for this **TypeInfo** object.
2. At "patch" time, we create a chain of initializations which should be performed; each entry in this chain will contain a pointer to each **TypeInfo** object which needs to be initialized, and a pointer to the corresponding initialized object. If there are be multiple initialized objects available, one is chosen arbitrarily. This algorithm applies when patching executables as well as shared libraries.
3. When "_main" is executed and when a shared library is loaded, all the **TypeInfo** initializations are performed before any other initialization code is executed. An initialization of a **TypeInfo** object is quick because we simply need to store a pointer to the initialized object within the **TypeInfo** object.

Patch cannot initialize **TypeInfo** objects during the patch phase itself, because of the method used by most linkers to implement tentative definitions. If a linker needs to allocate space for an uninitialized tentative definition, it will usually simply update the size of the uninitialized area, and the loader will be responsible for initializing this area to 0. Since there is no real image in the object file which contains the initialization data for uninitialized tentative definitions, there would be no way to "patch" it to a different value.

The scheme we have described does have some runtime cost, and this cost is the initialization of one word for each **TypeInfo** object allocated for non-polymorphic classes, and polymorphic classes without unique vtables. If a **TypeInfo** object is referenced in the main executable as well in shared libraries, it will be initialized multiple times.

4.4.2. Munch Implementation for Shared Libraries

The "munch" implementation remains similar to the non-shared library scheme. For the main executable, we create initialized definitions for all **TypeInfo** objects we find referenced. Since these are initialized definitions, there will be no runtime startup cost.

When "munching" a shared library, we cannot create initialized definitions for the **TypeInfo** objects we find referenced, since such a definition may already have been provided when munch was run on the main executable, and multiple definitions are not permitted. Instead, we generate runtime code to initialize the referenced **TypeInfo** when the shared library is loaded, and before any initialization code is executed. Note that as in the "patch" scheme, the same **TypeInfo** object may get initialized twice.

In order to minimize the runtime initialization cost, the **TypeInfo** object can contain a pointer to an initialized object created by "munch". All that needs to be done at runtime is the assignment of one pointer into the **TypeInfo** object.

5. Previous Work

This section describes the Dossier[3] solution to the type identification problem. Most of the commonly used C++ toolkits use some form of runtime type identification, and the Dossier scheme was developed for use in the InterViews toolkit.

5.1. Summary of Dossier Approach

In the Dossier scheme, a tool called **mkdossier** scans the source files of an application and generates a statically initialized Dossier structure for each class in the user program. This Dossier structure is accessed through a virtual *GetClassId* function which needs to be added to each class.

The paper goes on to propose some language extensions to simplify accessing the Dossier structure for a class:

- A predefined static member caller *dossier* would be added to each class.
- The syntax `typename::dossier` would be used to access the dossier of a type.
- The syntax `object.dossier` would be used to access the dossier of an object.
- The syntax `pointer->dossier` would be used to access the dossier of the class object being pointed to. If the class being pointed to was polymorphic then the dossier of the dynamic class being pointed to would be returned (i.e. *dossier* is a virtual static data member).

5.2. Comparison

The Dossier approach is similar to our approach in many respects:

- We both make available a unique value associated with a type, which allows users to perform comparisons to determine if two types are the same.
- Through this unique value, both approaches allow various kinds of functionality to be accessed by using member functions (e.g. the ability to traverse the ancestor list)

Although there are similarities in our approaches, there are some fundamental differences:

Support for all types

We would like to support type inquiry on all types, not just classes. We consider this an important consideration in exception handling since a user may throw any type (not just classes). We also anticipate that users of parameterized types may occasionally need to perform type inquiry operations on template *type* parameters (which can be of any type).

Language Syntax

Although both approaches implement a "virtual static data member" associated with a type, we use different syntaxes for accessing this member. The Dossier scheme uses the ::, ., and -> operators, whereas we propose introducing two new operators **stype** and **dtype**. The main reason for our choice is to provide a consistent syntax for accessing type information for all types, not just classes.

Extensibility

Regardless of how much information is made available for a type automatically (for example, a list of ancestor classes), there will always be some applications which need additional type information. Our paper discusses a method for extending the standard type information generated by the compiler (the developer will have to take steps to ensure that this additional type information gets associated with the type).

Implementation

The Dossier mechanism relies on a tool to process the sources for an application and generate Dossiers. Although the sources may be partitioned into multiple sets (to handle libraries), care must be taken to ensure that the same Dossier is not generated twice. Our paper describes some implementation schemes in which the compiler automatically generates the necessary information, and no additional processing is necessary.

6. Open Issues

The type identification mechanism presented in this paper provides a reasonably complete set of functionality for type related operations and handling type information. In developing this mechanism we discovered some issues that require further discussion.

● Non-Polymorphic Classes

Non-polymorphic classes inherently possess a certain inconsistency with regard to type identification. Although they form a subtype hierarchy in the same way as polymorphic classes, given a pointer to a non-polymorphic base class it is difficult to determine the true type identity of the actual object being dereferenced at runtime. There are various alternatives available:

- make all non-polymorphic classes polymorphic;
- make all non-polymorphic classes, except for "extern C" classes, polymorphic;
- introduce a pragma to control this;
- introduce a compiler option;

Each of these options has serious disadvantages.

● ptr_cast operator

The use of casts is usually unsafe. The idea of an alternate cast operator[7] that is supposed to be applied only when a legal conversion is possible, is attractive. This operator would raise an exception if applied incorrectly.

7. Conclusion

We have described a general type identification mechanism consisting of language extensions and library support. The language extensions introduced support a reasonably full set of type inquiries. The library class called **TypeInfo** has been introduced to allow access to compiler generated type information. While providing access to basic information about types, it also contains member functions which can be used to extend the compiler-generated type information. An implementation strategy has been presented to demonstrate that the proposed extensions can be implemented efficiently.

The proposed type identification mechanism should satisfy the requirements of application and class library developers for type identification, access to a subtype query mechanism, and run-time access to type information.

8. References

[1] Margaret A. Ellis, Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley, 1990

[2] Andrew Koenig and Bjarne Stroustrup, Exception Handling for C++, USENIX C++ Conference Proceedings, 1990

[3] John A. Interrante, Mark A. Linton, Runtime Access to Type Information in C++, USENIX C++ Conference Proceedings, 1990

[4] Keith E. Gorlen, An Object-Oriented Class Library for C++ Programs, Proceedings of the USENIX C++ Workshop, 1987

[5] Andre Weinand, Erich Gamma, and Rudolf Marty, ET++ - An Object-Oriented Application Framework in C++, ACM OOPSLA'88 Conference Proceedings, 1988

[6] Mark A. Linton, John M. Vlissides, and Paul R. Calder, Composing user interfaces with InterViews, Computer, 22(2):8-22, February 1989

[7] Bjarne Stroustrup, Personal communication

Representing Semantically Analyzed C++ Code with Reprise

David S. Rosenblum (dsr@research.att.com)

Alexander L. Wolf (wolf@research.att.com)

AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974–2070

Abstract

A prominent stumbling block in the spread of the C++ programming language has been a lack of programming and analysis tools to aid development and maintenance of C++ systems. One way to make the job of tool developers easier and to increase the quality of the tools they create is to factor out the common components of tools and provide the components as easily (re)used building blocks. Those building blocks include lexical, syntactic, and semantic analyzers, tailored database derivers, code annotators and instrumentors, and code generators. From these building blocks, tools such as structure browsers, data-flow analyzers, program/specification verifiers, metrics collectors, compilers, interpreters, and the like can be built more easily and cheaply. We believe that for C++ programming and analysis tools the most primitive building blocks are centered around a common representation of semantically analyzed C++ code.

In this paper we describe such a representation, called REPRISE *(REPResentation Including SEmantics)*. The conceptual model underlying REPRISE is based on the use of expressions to capture all semantic information about both the C++ language and code written in C++. The expressions can be viewed as forming a directed graph, where there is an explicit connection from each use of an entity to the declaration giving the semantics of that entity. We elaborate on this model, illustrate how various features of C++ are represented, discuss some categories of tools that would create and manipulate REPRISE representations, and briefly describe our current implementation. This paper is not intended to provide a complete definition of REPRISE. Rather, its purpose is to introduce at a high level the basic approach we are taking in representing C++ code.

1 Introduction

A prominent stumbling block in the spread of the C++ programming language has been a lack of programming and analysis tools to aid development and maintenance of C++ systems. The development of such tools has been, and continues to be, a daunting prospect. One reason for this is that the language itself has been evolving, making it risky to invest much effort in the development of language-specific tools; fortunately, this situation appears to be improving. Another reason is that the language is inherently complex, both in its syntax and semantics. The developer of anything more than the most trivial of tools is faced with the prospect of having to devote a significant portion of their tool—and time—to dealing with this complexity.

One way to make the job of tool developers easier and to increase the quality of the tools they create is to factor out the common components of tools and provide those components as easily (re)used building blocks. We believe that for C++ programming and analysis tools the most primitive building blocks are centered around a common representation of semantically analyzed C++ code.

Consider, for example, the design of the C++ Information Abstractor (CIA++) [9]. CIA++ is a tool that constructs a database of information about the non-local entities in a C++ program. A variety of display and analysis tools make use of this database to provide information to developers. Thus, CIA++ is a building-block tool in the sense that it offers a common service, namely the specialized filtering and structuring of information about C++ code, to a number of other tools. The current version of CIA++ was built by painful modification of *cfront* [1], which is a tool that performs lexical, syntactic, and semantic analysis of C++ code, as well as a translation of the C++ code into C. Relatively simple modifications included the careful removal of all code performing the actual generation of C code. The difficult modifications stemmed from the manner in which the internal data structures and code are organized in *cfront.* Specifically, the collection and representation of semantic information is spread out across the code. Indeed, certain semantic information is "thrown away" at various times during the translation process. While this organization might make sense for translation of C++ to C, it made the design of CIA++ very complex. Furthermore, to maintain compatibility with the "official" *cfront,* updates to the official version must be carefully incorporated into CIA++. Had there instead been available a data structure representing the semantics of C++ code (and, of course, a tool to generate such a data structure) the developers of CIA++ could have written a relatively simple tool to derive CIA++ databases directly from the data structure representation. Thus, the design of CIA++ would have been greatly simplified, the time required to implement it would have been drastically reduced, and the need to track updates to *cfront* would have been avoided.

We have developed REPRISE, a representation for semantically analyzed C++ code.[1] REPRISE can serve as the basic data structure for the building blocks of a wide variety of tools. Those building blocks include lexical, syntactic, and semantic analyzers, tailored database derivers (e.g., CIA++), code annotators and instrumentors, and code generators. From these building blocks, tools such as structure browsers, data-flow analyzers, program/specification verifiers, metrics collectors, compilers, interpreters, and the like can be built more easily and cheaply.[2] Factoring out the primitive components in this manner would free tool developers to concentrate on the unique, critical aspects of their tools. An additional benefit of this approach is that tools operating on the same C++ code can share the REPRISE representation of that code, resulting in a significant savings in both space and time. Given our experience in defining, building, and using Ada programming and analysis tools (e.g., [5, 14, 15, 20]) based on the DIANA [7] and PARIS [8] representations, we believe that this approach to the development of tools for C++ is a viable one to consider.

We begin in Section 2 by describing the model upon which the representation is based. In Section 3 we sketch, through examples, how C++ code is actually represented in REPRISE. In Section 4 we discuss some categories of tools that would create and manipulate REPRISE representations. We conclude in Section 5 with a brief description of our current implementation.

Note that this paper is not intended to provide a complete definition of REPRISE. Rather, its purpose is to introduce at a high level the basic approach we are taking in representing C++ code.

[1] REPRISE is an acronym for *REPResentation Including SEmantics.* We intend this name to evoke a feeling of reuse of representations, as in the musical term *reprise,* "a repetition of a phrase or verse" [11].

[2] Whether something is viewed as a tool or as a building block for tools is, of course, a matter of perspective. For all intents and purposes, a building block is a kind of tool, so we do not distinguish between building blocks and tools in the remainder of this paper.

2 The Representation Model

As mentioned above, we imagine that a representation for semantically analyzed C++ code could be used profitably by a wide variety of tools. The accommodation of such variety requires that the representation exhibit the following characteristics:

- *Primacy of semantics.* The form of the representation must be driven by the semantics of the language constructs, not by their syntax, since it is primarily at the semantic level that sophisticated tools need to manipulate information about C++ code.[3]

- *Regularity of form.* The manipulations needed for basic processing of the representation, such as traversal, must be straightforward to understand and implement. Regularity of form reduces the need for tools to contain complicated, special-case code.

- *Minimization of speciality.* Different tools may want to treat portions of C++ code differently. These necessary and desirable idiosyncrasies, however, must not be embedded in the form of the representation. Rather, they should be reflected in the tools themselves. We cannot expect to be able to anticipate the needs of all tools and, moreover, those needs may be in conflict.

- *Evolvability of representation.* Although the language definition is stabilizing, there continue to be proposals for changes (e.g., for templates [16] and exception handling [10]). It is important, therefore, that the representation can be easily evolved along with the language. To facilitate this, the representation must capture more than just the surface-level, user-visible semantics. It must also effectively capture the basic fabric of the language—that is, the semantics of how the language is put together.

- *Uniformity of representation.* The basic fabric of C++ must be represented in the same way as user-written code. Indeed, cognizance that a particular entity is primitive (i.e., a component of the basic fabric) or not is something that should be left up to individual tools. Uniformity of representation, like regularity of form, reduces the need for special-case code in tools. Moreover, it allows tools to be constructed in such a way that they can more easily evolve along with the language and representation.

These characteristics form recurring themes throughout the design of REPRISE.

The conceptual model underlying REPRISE can be viewed from two equivalent perspectives. The first is as an expression language in which all semantic information about both the C++ language and code written in C++ is uniformly represented as the application of operators to arguments. For example, the expression `F != 0`, where `F` is a pointer variable, is represented as an application of the language-defined inequality operator `!=` to the object `F` and the literal `0`. A more interesting example is a representation for a construct like the *if-statement,* where the language-defined operator `if` is applied to two arguments, one an expression representing a condition and the other an expression representing the statement to execute if the condition is true. Even declarations are represented as expressions, as discussed in Section 3.

A second way of viewing REPRISE, and the one that we tend to use most often, is as a directed graph. Unlike a traditional abstract syntax tree, a REPRISE graph explicitly captures semantic information by connecting, through edges, entity uses with entity declarations. To do so, a REPRISE graph, which could well be called an "abstract semantics graph", employs two kinds of nodes and two kinds of directed edges. The first kind of node is used to represent expressions (i.e., applications of operators to arguments) and the second kind is used to represent literals (e.g., identifiers, numbers, or strings). Expression nodes have one or more children. The first child is always an expression representing the declaration, and hence semantics, of the operator being applied, while the other children are the arguments to the operator. The two kinds of edges are used to distinguish between the following two situations: *i*) an expression node having a child that is the result of a previously evaluated expression, and *ii*) an expression node having a child that is an expression to be evaluated

[3]Note that this characteristic can lead to easy accommodation of graphics-based tools (i.e., those that present C++ code to users through an iconic, rather than textual, syntax).

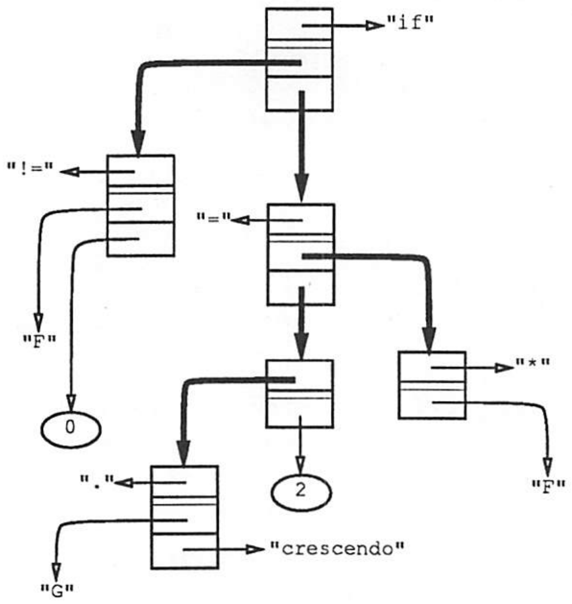

Figure 1: REPRISE Representation of an if-statement.

as part of the parent expression's evaluation. We call the first kind of edge a *reference edge,* since the parent expression is actually referring to a previously determined result, and call the second kind an *evaluation edge,* since the child expression is evaluated as part of the application of the parent's operator. Reference edges are also used to refer to children that are literal nodes. The need to distinguish edges is a rather subtle issue explored further in Section 3. Fortunately, while the distinction must be evident in the representation, it can in practice be ignored by most tools.

Figure 1 depicts a portion of a REPRISE graph representing the code fragment

```
if (F != 0) *F = G.crescendo(2);
```

where rectangles are expression nodes, ovals are literal nodes, dark arrows are evaluation edges, and light arrows are reference edges. The connections between this code fragment and the declarations for entities `if`, `!=`, `*`, `=`, `.`, `crescendo`, `F`, and `G` are implied by the pictorial abbreviation →`"entity"`. (Many of the abbreviations appearing in this figure are expanded in Figure 3.) Although it is not evident from the depiction, all occurrences of the same abbreviation in this figure denote the same entity declaration. Thus, both edges →`"F"` actually terminate at the same node, the root of the subgraph representing the declaration of `F`. It is this kind of sharing that makes REPRISE a non-tree graph. Indeed, a REPRISE graph may contain cycles.

The connections between uses and declarations drawn by edges are what capture the semantics of the code being represented. For instance, looking at the expression node in Figure 1 that represents the selection of member `crescendo` from object `G`, we see that it is an application of the language-defined "dot" operator (i.e., the operator whose declaration is at the other end of the reference edge[4]) to a particular object (the one whose declaration is at the other end of the reference edge) and an expression (at the other end of the evaluation edge). A more extensive example is given in the next section.

While a REPRISE graph clearly is not an abstract syntax tree, one easily can see where the traditional abstract syntax tree resides, as a subgraph, within it. The existence of this subgraph is important, since tree-based algorithms are generally more efficient than their graph-based counter-

[4] A reference edge is used here because functions in C++ are first declared (the declaration expression is evaluated) and then applied (referenced) in some number of other places, such as this one, in the code.

parts; the abstract syntax tree subgraph can be used where appropriate in manipulating a REPRISE representation.

Another subgraph of interest in a REPRISE representation is what we refer to as the *core graph.* The core graph comprises all the nodes and edges used to represent the semantics of C++.[5] Thus, the core graph is totally self contained; no edges leave the core graph. Included in the core graph are representations of such things as the declarations of the operators `if` and `switch`, as well as declarations for the types `int` and `float`. Also included are even more primitive operators and types, such as `%list` and `%numeric`,[6] that are not explicit in any user-written code, but are employed both in the representation of the primitive operators and types themselves and in the representation of user-written code. Conceptually, the core graph forms part of every REPRISE representation. Because it is self contained, however, the core graph can be physically shared among those representations.

The core graph provides one view of the C++ primitive semantics. Taking the expression-language perspective, the primitive semantics can be viewed as a set of expressions declaring the primitive types and operators. In other words, C++ can be described as a particular set of declared entities, where user-written code makes use of those declared entities. The advantage of this perspective is that it becomes evident what the relationships, and "non-relationships", are among the primitive entities. Moreover, additions to the language, such as those proposed for exception handling, amount to the declaration of some additional entities.

Let us relate the model underlying REPRISE to the characteristics listed at the beginning of this section. The primacy of semantics is reflected in the explicit connection in the representation from each use of an entity to the declaration giving the semantics of that entity. Indeed, it is these semantic connections that make the representation much more than a typical abstract syntax tree. Representing all aspects of C++ code as expressions and employing only two kinds of nodes and two kinds of edges leads to both a very regular form for representations and a minimum of speciality. Nevertheless, accommodating the special needs of specific tools, such as keeping track of exercised branches and statements for test-coverage analysis, is straightforward, as discussed in Section 4. Finally, the REPRISE core graph captures the complete primitive semantics of C++ and, moreover, does so in the same form as user-written code. Having the primitive semantics in this form is useful in evolving the representation to accommodate new language features, since it is a particularly malleable data structure.

3 Representing C++ Code with Reprise

In this section we describe how C++ code is represented as REPRISE graphs. The terminology we use generally follows that of the C++ reference manual [6]. For purposes of illustration, we use the REPRISE representation of the C+ program of Figure 2, a simple example involving a class called `ical`. The REPRISE representation of this program is shown in Figure 3, except that core graph entities are represented by the pictorial abbreviation →`"entity"`, as before. Note that the source program of Figure 2 includes the if-statement whose REPRISE representation is depicted in Figure 1; thus, Figure 3 also illustrates how the representation of the if-statement fits into the larger context of a complete C++ program.

The entities appearing in C++ user-written code fall into three broad categories: types, declarations, and statements. We illustrate each of these categories with representative examples extracted from Figure 3 and then conclude with a discussion of some of the conventions we use in representing C++ code with REPRISE.

3.1 Representation of Types

There are three aspects to the representation of types in a REPRISE graph: the hierarchy of predefined C++ types, a collection of operators called *type constructors* for representing the formation of user-

[5] Of course, the most primitive semantics, namely operator application and argument evaluation, are not explicitly represented in the graph, but are assumed to be "understood" by all tools.

[6] We prepend the character "`%`" to the names of primitive entities not available for use in user-written C++ code.

```
class ical {
private:
        int p, f;
public:
        ical crescendo(int const c);
        ical() { p = f = 1; }
};

void main()
{
        ical* F = 0;
        ical G;
        if (F != 0) *F = G.crescendo(2);
}
```

Figure 2: A Sample C++ Program.

defined types, and a collection of operators called *type modifiers* for representing additional attributes of types.

Figure 4 depicts the hierarchy of predefined types in C++. As shown in the figure, this hierarchy is a subtype relationship. The subtype relationship is defined inductively in terms of the functions that are defined for each type T. In particular, the functions that are *applicable* to objects of type T include *i*) zero or more functions that are defined explicitly for T, plus *ii*) the functions applicable to object's of T's supertype. In addition, the language defines certain implicit type promotions and conversions that allow the functions for one type to be applied to objects of another type. In other words, if a function applied to an object of type T is not defined for type T, then either the function must be defined for some supertype of T, or else there must be an implicit type promotion or conversion defined by the language from type T to type T' such that the function is defined for type T'.

The fundamental types of C++ (`char`, `int`, etc.) appear at the "leaves" of the subtype hierarchy and are shown in boldface in Figure 4. The remainder of the hierarchy comprises several *meta-types* (denoted by a leading "`%`"), which are never used explicitly within user-written code. Figure 3 illustrates references to the fundamental types `int` and `void`, as well as references to the meta-types `%func`, `%class`, and `%pointer`.

The utility of the subtype hierarchy can be appreciated by considering the fundamental type `int`. The language predefines several functions for `int`, namely the arithmetic operators (unary and binary + and -, *, /, and %), bitwise operators (~, &, |, and ^), shift operators (<< and >>), and relational operators (<, >, <=, >=, ==, and !=). As shown in Figure 4, `int` is a subtype of `%numeric`. The language predefines several functions for `%numeric`, namely the ternary conditional operator (?:), increment operators (++), decrement operators (--), and logical operators (!, &&, and ||). Thus, because `int` is a subtype of `%numeric`, all of the operators defined for `%numeric` are applicable to objects of type `int`.

A type constructor represents the formation of a user-defined type from one or more existing "ingredient" types,[7] while the type modifiers qualify a type as being either a constant, volatile, or reference type. Each user-defined type is a subtype of one of the meta-types shown in boxes in Figure 4. In order to represent the subtyping semantics described above, one argument to each type constructor is a reference to the type's supertype in the subtype hierarchy. The other arguments to a type constructor represent other attributes of the constructed type, including the ingredient type(s) from which it is constructed (such as the base types of a class). The sole argument to a type modifier is a reference to the modified type. Figure 3 illustrates the use of the type constructors `class`, `%function`, and `*`, as well as the type modifier `const`.

The operator `class` is used to represent the definition of a class, structure, or union. All non-

[7]User-defined types are referred to as "derived types" in Section 3.6.2 of the C++ reference manual [6].

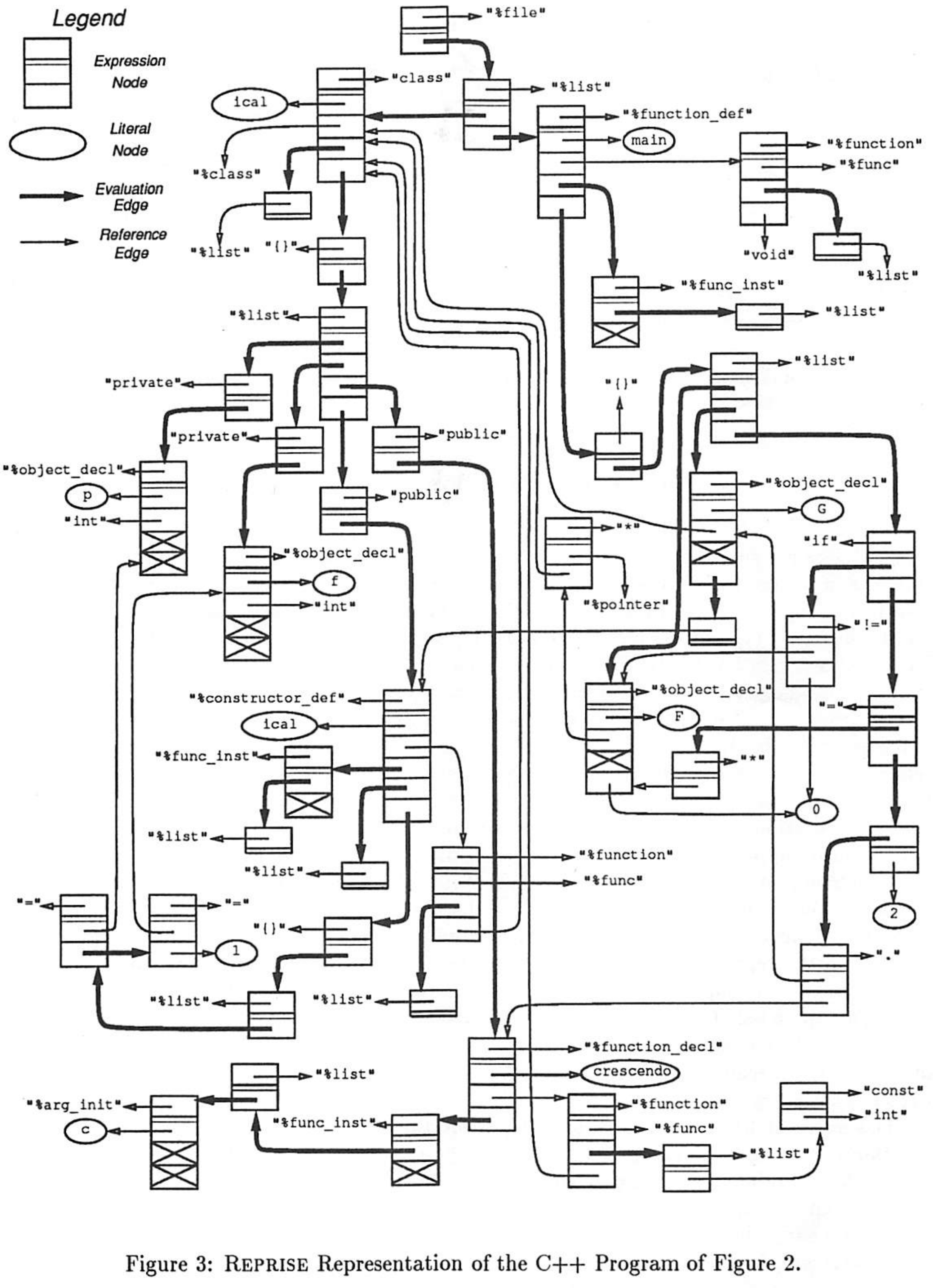

Figure 3: REPRISE Representation of the C++ Program of Figure 2.

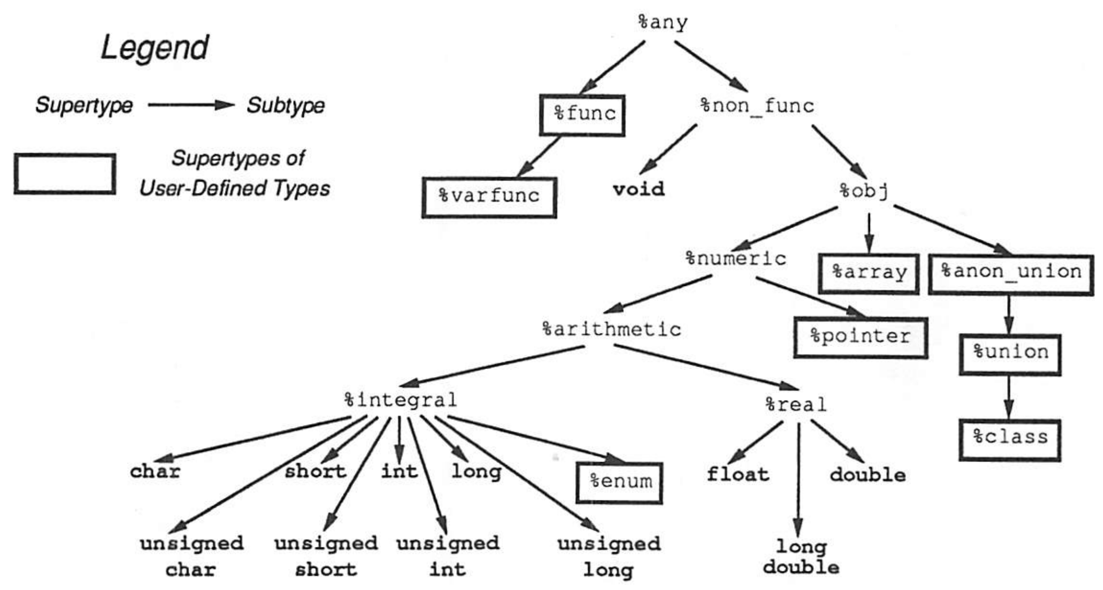

Figure 4: Subtype Hierarchy of Predefined C++ Types.

union classes are subtypes of the meta-type `%class`. Anonymous unions are subtypes of the meta-type `%anon_union`, while all other unions are subtypes of the meta-type `%union` (see Figure 4). Note that from the perspective of the subtype relationship, a structure is semantically the same as a class and is thus a subtype of the meta-type `%class`. Non-union classes are further related according to an inheritance relationship that is specified explicitly in the user-written code, using the inheritance syntax of C++. The semantics of the inheritance relationship is quite different from the semantics of the subtype relationship, and hence the two relationships are represented in different ways in a REPRISE graph. In particular, given a class *C* and a class *D* derived from *C*, the inheritance relationship between the two classes is represented by a reference edge from the representation of *D* to the representation of *C*. On the other hand, both classes are subtypes of `%class`, and therefore the representations of the classes are connected by reference edges to the representation of `%class`. If there also happens to be a subtype relationship between *C* and *D*, this fact can be determined from their inheritance relationship and from their definitions. The differences in the semantics of the inheritance and subtype relationships are discussed in detail by Moss and Wolf [12].

Figure 5 illustrates the use of the operator `class`; it contains the portion of the graph of Figure 3 devoted to the representation of class `ical`. As shown in the figure, `class` takes a name as its first argument, the appropriate supertype as its second argument, a list of base classes as its third argument, and a list of member declarations as its fourth argument. Both lists are represented by an application of the operator `%list`, which is defined in the core graph and can take a variable number of arguments. Because class `ical` has no base classes, the second argument to this application of `class` is an empty list.

The operator `%function` is used to represent the type of a function in the representation of function declarations, function definitions, and pointer-to-function types. Figure 6 illustrates the use of the operator `%function`; it contains the portion of Figure 3 devoted to representing the type of member function `crescendo` of class `ical`. Figure 6 also illustrates the use of the type modifier `const`, which simply qualifies its argument as being a constant type. The first argument to `%function` is a reference to the supertype of the function type, the second argument is a list of argument types, and the third argument is a reference to the return type. The supertype of a function type having a fixed number of arguments is the meta-type `%func`, while the supertype of a function type having a variable number of arguments is the meta-type `%varfunc`. Note that

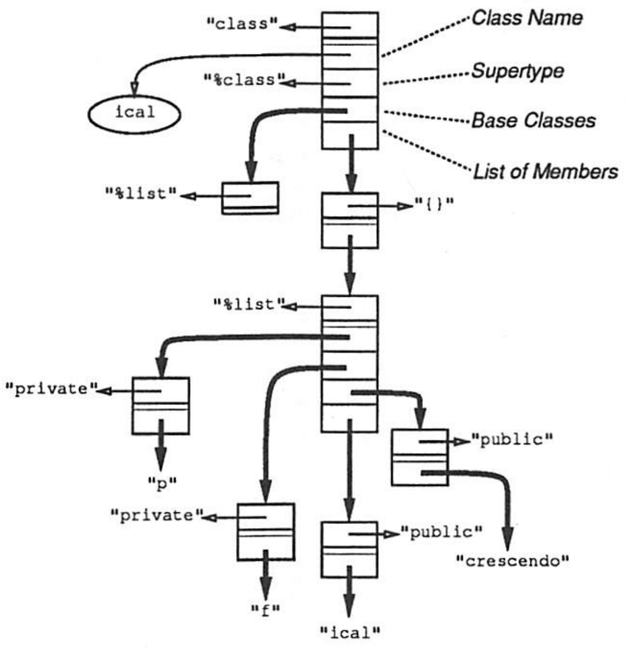

Figure 5: REPRISE Representation of Class **ical**, Extracted from Figure 3.

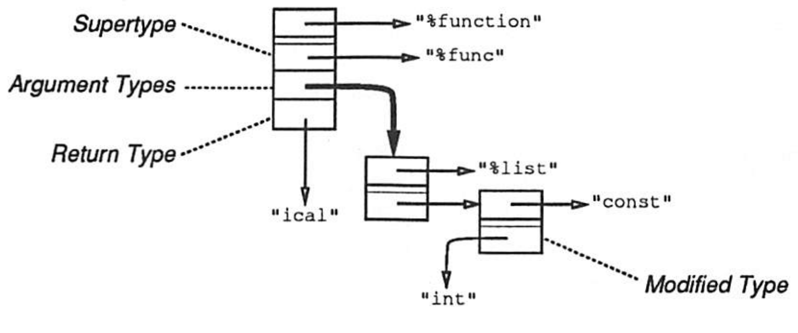

Figure 6: REPRISE Representation of the Type of Function **crescendo**, Extracted from Figure 3.

argument names and argument initializers are not semantically part of the type of a function; for instance, two functions that have the same set of argument and return types, but different sets of argument names and initializers, could both be assigned to a variable of the appropriate pointer-to-function type (which may have been declared with still another set of names and initializers). Therefore, argument names and initializers are associated with the representation of each declaration whose type involves a function type (e.g., a function, or a variable that is a pointer to a function), instead of with the representation of the function type itself (see Section 3.2).

Two additional things should be noted about the representation of types in a REPRISE graph. First, supertypes and ingredient types are existing types from which new types are built; thus, when supertypes and ingredient types are specified as arguments to a type constructor, they are specified with reference edges. Second, when newly defined components of a type (such as a list of class members or function argument types) are specified as arguments to a type constructor, they are specified with evaluation edges.

3.2 Representation of Declarations

C++ declarations are represented by expressions involving operators called *declaration constructors.* A declaration constructor represents the declaration of a variable, function, or typedef. Figure 3 illustrates the use of the declaration constructor for variables (`%object_decl`) and three of the declaration constructors for functions (`%function_decl`, `%function_def`, and `%constructor_def`). Note that class members are represented simply as variables or functions that are defined within classes. In addition to the declaration constructors, several *declaration modifiers* are used to represent various attributes of declarations, such as access specifiers (e.g., `private`), storage class specifiers (e.g., `static`), function specifiers (e.g., `friend`), and linkage specifications (e.g., `extern "C"`).

All declarations in C++ code have an associated *scope* within which they are visible. Scopes are represented in a REPRISE graph by several operators that are defined in the core graph. In particular, a file scope is delimited by an application of the operator `%file`, as shown in Figure 3. The scope of a function argument or statement label is delimited by an application of one of the declaration constructors for functions. The scope of a variable declared in a for-loop header is delimited by an application of the operator `for`. All other scopes are delimited by the operator {},[8] which is used to denote the scope of a declaration in a compound statement (including the outermost scope of a function body) and the scope of a class member.[9]

3.2.1 Variable Declarations

A variable declaration is represented by an application of the operator `%object_decl`. Figure 7 depicts the portion of the graph of Figure 3 devoted to the representation of data member `p` of class `ical`. The first argument to `%object_decl` is the name of the declared object. The second argument is a reference to the type of the object. The third argument is the list of name/initializer pairs for the type; in this case, the third argument is null since the type does not involve a function type. The fourth argument is an initializer for the object, which in Figure 7 is null because no initializer is specified for `p`.

The `%object_decl` expression of Figure 7 by itself simply represents the declaration of a variable called `p`. However, since `p` is a member of class `ical`, the `%object_decl` expression appears as the argument to an instance of the declaration modifier `private`, which represents the fact that `p` is a private class member. In addition to the declaration modifier `private`, there are also declaration modifiers `public` (see Figure 3) and `protected` for representing access specifiers of class members.

The representation of the declaration of `G` in Figure 3 illustrates how initialization of class objects is represented. Even though no initializer is specified explicitly for this variable in the program of Figure 2, semantically it is initialized by the constructor that is defined for class `ical`. Thus, the fourth argument to the `%object_decl` expression is an expression representing a call to this constructor.

[8] The operator {} is pronounced "Curly".

[9] Note that the access specifiers `protected` and `public` in effect *extend* the scope of a class member.

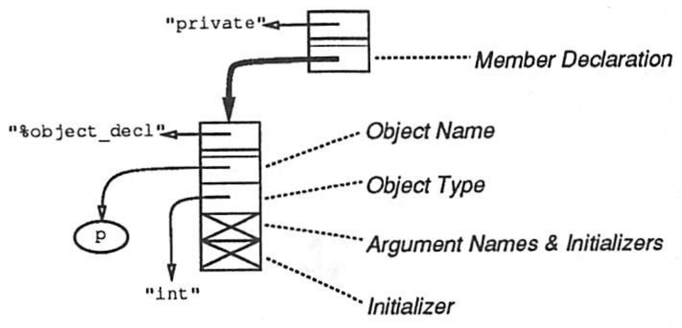

Figure 7: REPRISE Representation of Data Member p of Class `ical`, Extracted from Figure 3.

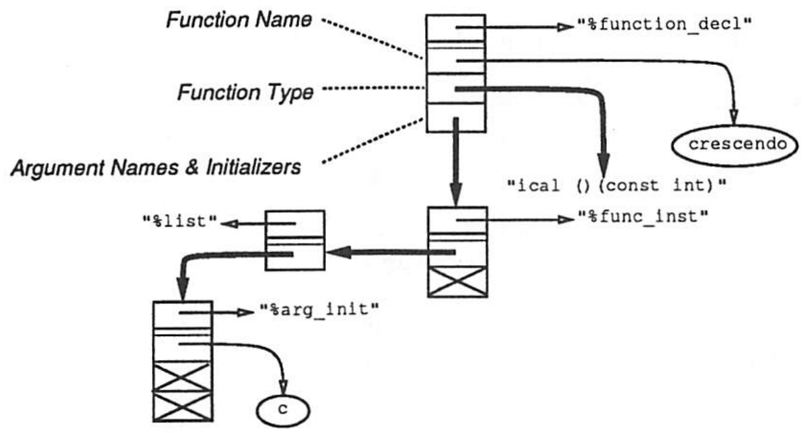

Figure 8: REPRISE Representation of Member Function **crescendo**, Extracted from Figure 3.

As is done in the representation of types, reference edges and evaluation edges are used to distinguish, respectively, between the existing entities in a REPRISE graph upon which a declaration depends (such as the type of the declared entity) and those entities that are defined specifically for a declaration (such as the initializer of an object, or the argument name/initializer pairs and body of a function).

3.2.2 Function Declarations and Definitions

A function is represented by an application of either the operator **%function_decl** or the operator **%function_def**. The former is used to represent functions declared with a header but no body, while the latter is used to represent functions that are declared with their bodies. The only difference between the two is that the latter has an extra argument for representing the function body.

Figure 8 contains the portion of the graph of Figure 3 devoted to the representation of member function **crescendo** of class **ical** and illustrates the use of the operator **%function_decl**. The first argument to **%function_decl** is the name of the declared function. The second argument is a reference to the type of the function; the representation of the type of **crescendo** is shown in Figure 6. The third argument represents the name/initializer pairs for this particular use of the function type. We call such a particular use a *function-type instantiation*. As mentioned above, the name/initializer pairs are associated with the function declaration rather than the function type because semantically they are not attributes of the function type.

As shown in the figure, the operator `%func_inst` is used to represent a function type instantiation. The first argument to `%func_inst` is the list of name/initializer pairs for the arguments of the function type. The second argument is the `%func_inst` expression for the return type, which in this case is null since the return type does not instantiate a function type. The operator `%arg_init` is used to represent each name/initializer pair. The first argument to `%arg_init` is the name of the argument, the second argument is the `%func_inst` expression for its type, and the third argument is its initializer. The second and third arguments to `%arg_init` are both null in this case since `c` has no initializer and its type does not instantiate a function type.

Class constructors and destructors are special kinds of functions and are represented by applications of the operators `%constructor_decl`, `%constructor_def`, `%destructor_decl`, and `%destructor_def`. The semantics of the arguments to the operator `%constructor_decl` are identical to those of `%function_decl`. The operator `%constructor_def` is similar to `%function_def` except that it has an additional argument (its fourth argument) for representing the optional list of initializers for members and base classes; the use of this operator is illustrated in Figure 3, which shows the fourth argument as being an empty list because there are no initializers specified for the constructor of class `ical`. The operators for representing destructors differ from those for representing "normal" functions only in their lack of an argument for representing function argument/initializer pairs, since destructors do not take arguments.

Overloaded functions are represented using the declaration constructors for functions in the same manner as is illustrated in Figure 3 for functions that are not overloaded. There is nothing special about the representation of an overloaded function other than the fact that multiple function declarations with the same name can can appear in a REPRISE graph. However, unlike a simple abstract syntax tree representation of C++ code, a REPRISE graph contains no ambiguities as to which function is being called in a reference to an overloaded function, because each such reference is resolved in the form of a semantic connection between the reference and its matching declaration.

C++ defines several functions on its fundamental types, such as the operator `+` that takes two `int` arguments and returns an `int`. The declarations of these predefined functions are represented by `%function_decl` expressions in the core graph. Certain predefined functions, such as the pointer dereferencing operator `*`, are polymorphic, since they are defined once for a category of types. A full discussion of the representation and use of polymorphism in REPRISE is beyond the scope of this paper.

3.3 Representation of Statements

Representation of C++ statements in REPRISE is straightforward. C++ expression statements fit naturally with REPRISE's expression-based model and are thus represented simply by nested applications of operators. Each of the remaining kinds of C++ statements is represented by a REPRISE operator whose name is the same as that of the statement (`if`, `switch`, `continue`, etc.). Figure 3 illustrates the representation of three kinds of statements—expression statements, if-statements, and compound statements. The operator `if` is used to represent if-statements and is described in Section 2. The operator {} represents compound statements, simply taking a list of declarations and statements as its sole argument.

3.4 Conventions

Our method of representation adheres to several informal conventions. For example,

- the first argument to each declaration constructor specifies the name of the declared entity;
- the second argument to each declaration constructor specifies the type of the declared entity; and
- an operator argument that is an empty list is represented by a `%list` expression having zero arguments, rather than by the null value.

These and other such conventions add to the uniformity of the representation, further simplifying the job of implementing REPRISE-based programming and analysis tools.

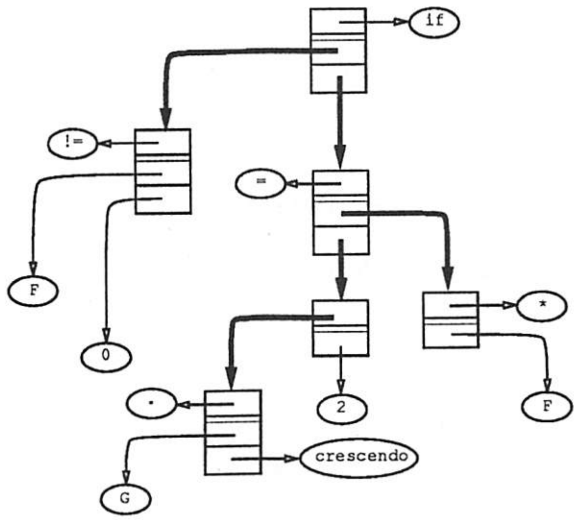

Figure 9: "Pre-semantic" Form of the REPRISE Representation in Figure 1.

4 Reprise and Tools

As discussed in Section 1, the purpose of REPRISE is to serve as a common data structure for programming and analysis tools that depend upon semantic information about C++ code. The previous two sections concentrate on the representation itself. Here we turn attention to the tools that would make use of the representation. There appear to be at least four categories of such tools, each of which is discussed below. For purposes of exposition, the categories are discussed separately, but of course hybrid tools are also possible.

4.1 Generative Tools

Generative tools create REPRISE representations. The most obvious generative tool is what traditionally serves as the front end of a compiler, performing lexical, syntactic, and semantic analysis. The input would be C++ source text and the output would be the REPRISE representation of that source text. One way that this tool could work (in fact, the way that such tools work for the PARIS representation of Ada) is to first generate the abstract syntax tree subgraph through lexical and syntactic analysis, and then modify that subgraph through semantic analysis to capture the semantic connections between entity uses and entity declarations. Figure 9 shows the so-called "pre-semantic" form of the representation shown in Figure 1. In essence, the "post-semantic" form of Figure 1 differs from the pre-semantic form in that certain literal nodes are replaced with semantic connections to the declarations of the referenced entities. Note that as a consequence of the overloading and scoping rules of C++, there is likely to be a many-to-one mapping of literals to declarations in the pre-semantic form. The direct capture of the semantic connections in (post-semantic) REPRISE, however, eliminates any possible ambiguity.

Less traditional generative tools would be C++ semantics-directed editors[10] and graphics-based editors, where the textual syntax of C++ has been replaced with an iconic syntax. While these tools would not take actual C++ source text as input, they would still produce REPRISE representations that could be manipulated by the other categories of tools.

[10] Today's "syntax-directed" editors generally perform semantic checks as well, and thus it is really a disservice to continue calling them syntax-directed editors rather than semantics-directed editors.

4.2 Deriver Tools

Deriver tools create specialized representations of C++ code, using REPRISE representations of the code as their input. These specialized representations are tailored to the needs of particular tools or set of tools. A specialized representation might, for instance, contain a subset of the information contained in a REPRISE representation or provide a different structure to the information appropriate to a particular kind of processing. CIA++, discussed in the introduction, is an excellent candidate for a deriver tool. Of course, specialized representations do not have to replace the REPRISE representations from which they were derived, but can be used in conjunction with those REPRISE representations as well. For instance, a tool that creates an index of identifier names, for fast access to declarations in a REPRISE graph, can be thought of as a deriver tool.

4.3 Annotation and Instrumentation Tools

Annotation and instrumentation tools "decorate" the REPRISE representation of C++ code with specialized information. Examples of tools that produce or make use of such decorations are test-coverage tools, performance analyzers, debuggers, and tools that produce embedded constraint-checking code from formal specifications (e.g., APP [13]). A test-coverage tool, for example, might work as follows: Given a piece of code and some test input, the tool tries to make a determination of which branches would be taken in the code and which statements would be executed. For each branch and statement in the code, the tool keeps track of which test input, if any, exercised (i.e., covered) that branch or statement.

The key issue raised by annotation and instrumentation tools is whether specialized, supplementary information, such as test coverage, can be associated with a REPRISE representation in an unobtrusive way—that is, without affecting tools not concerned with the information. To some extent, this is a question of how REPRISE as an abstraction is actually implemented, since the choice of implementation technique can have a significant impact on this issue [18]. In general, however, we note that REPRISE lends itself to an approach in which nodes and edges can be uniquely identified, and that those identities can be used as keys for auxiliary data structures. The values associated with those keys, and hence with nodes and edges, make up the supplementary information relevant to particular tools. Thus, in the case of the test-coverage tool, information about which test inputs exercise which branch or statement can be captured in a separate data structure known only to the test-coverage tool. The connection between that data structure and the C++ code is made by the unique identity of nodes and edges representing branches and statements in a REPRISE graph.

An interesting tool to consider in this category is the preprocessor *cpp* [1]. This tool interprets special statements, called *preprocessor directives,* that appear in files containing C++ code. The preprocessor uses the directives to determine what C++ code to pass on to other tools for processing. There are some tools, such as CIA++, that make use of information contained in preprocessor directives, and therefore it is important to be able to retain this information along with the REPRISE representation of the actual C++ code. The natural way to do this is to have an auxiliary data structure (perhaps created by a version of *cpp*) that serves this purpose. An important advantage of this approach is that the connections between preprocessor directives and C++ code can be made at a semantic level, not simply a syntactic one.

4.4 "Vanilla" Tools

Generally, tools that do not create REPRISE representations, derive specialized representations, or annotate or instrument C++ code with specialized information, use REPRISE representations directly and as is. In other words, the REPRISE representation contains all the information the tools would need and in a form appropriate to their tasks. Tools that would fall into this category include pretty printers, metrics collectors, data-flow analyzers, inheritance-hierarchy displayers, code generators, and the like.

5 Conclusion

The efficacy of programming and analysis tools using and sharing graph-based representations of semantically analyzed code has already been demonstrated for languages other than C++. The Arcadia environment [17], for example, has a full arsenal of concurrency analysis [2, 21], interface analysis [20], testing [4], and interpretation [22] tools, all based on such representations of Ada and Ada-like code. REPRISE represents an application of this technology to C++ programming environments.

To date, we have implemented the REPRISE data structures as a library of C++ classes. The classes are built on top of a persistence library called *Persi* [19], which supports long-term storage of C++ objects and shared concurrent access to those objects. We have built an enhanced version of *cfront* called *rfront*, which generates pre-semantic REPRISE graphs from C++ source code. We have also implemented a variety of tools that manipulate REPRISE representations, including a prototype name resolver that transforms pre-semantic graphs into post-semantic graphs.

We believe that REPRISE provides an excellent mechanism for constructing C++ environments, for simplifying development of C++ programming and analysis tools, and for increasing tool quality. In other words, we see REPRISE increasing the tempo at which C++ programming and analysis tools are developed, and orchestrating an harmonious interaction among them.

Acknowledgements

The conceptual model underlying the design of REPRISE was strongly influenced by IRIS [3]. IRIS is being used in the Arcadia software development environment research project [17] as the basis for representations of code written in a number of languages, including Ada, PIC/ADL [20], and GDL [5].

References

[1] AT&T. *UNIX® System V C++ Language System Release Notes*, release 2.1 edition, 1990. Select Code 307-160.

[2] G.S. Avrunin, L.K. Dillon, and J.C. Wileden. Experiments with constrained expression analysis of concurrent software systems. In *Proceedings of the SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 124–130, Key West, FL, December 1989. ACM SIGSOFT.

[3] D.A. Baker, D.A. Fisher, and J.C. Shultis. The gardens of Iris. Technical report, Incremental Systems Corporation, Pittsburgh, PA, 1988.

[4] L.A. Clarke, D.J. Richardson, and S.J. Zeil. TEAM: A support environment for testing, evaluation, and analysis. In *Proceedings of SIGSOFT '88: Third Symposium on Software Development Environments*, pages 153–162, Boston, MA, November 1988. ACM SIGSOFT. Appears in *ACM SIGSOFT Notes*, Vol. 13, No. 5.

[5] L.A. Clarke, J.C. Wileden, and A.L. Wolf. GRAPHITE: A meta-tool for Ada environment development. In *Proceedings of the Second International Conference on Ada Applications and Environments*, pages 81–90, Miami Beach, FL, April 1986. IEEE Computer Society.

[6] M.A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[7] A. Evans, K.J. Butler, G. Goos, and W.A. Wulf. *DIANA Reference Manual, Revision 3.* Tartan Laboratories, Inc., Pittsburgh, PA, 1983.

[8] K. Forester, I. Shy, and S. Zeil. PARIS operators: An Arcadia perspective. Technical Report Arcadia Document UCI-88-01, Department of Information and Computer Science, University of California at Irvine, 1988.

[9] J.E. Grass and Y. Chen. The C++ information abstractor. In *Proceedings of the Second C++ Conference*, San Francisco, CA, April 1990. USENIX.

[10] A.R. Koenig and B. Stroustrup. Exception handling for C++ (revised). In *Proceedings of the Second C++ Conference*, San Francisco, CA, April 1990. USENIX.

[11] W. Morris, editor. *The American Heritage Dictionary of the English Language*. Houghton Mifflin Company, Boston, MA, 1975.

[12] J.E.B. Moss and A.L. Wolf. Toward principles of inheritance and subtyping in programming languages. (available as AT&T Bell Laboratories Technical Memorandum 59113-881010-12TM), October 1988.

[13] D.S. Rosenblum. APP: An annotation preprocessor for creating self-checking C programs. (in preparation).

[14] D.S. Rosenblum. A methodology for the design of Ada transformation tools in a DIANA environment. *IEEE Software*, 2(2):24–33, March 1985.

[15] S. Sankar, D.S. Rosenblum, and R.B. Neff. An implementation of Anna. In *Ada in Use: Proceedings of the Ada International Conference*, pages 285–296, Paris, France, May 1985. Cambridge University Press.

[16] B. Stroustrup. Parameterized types for C++. In *Proceedings of the C++ Conference*, pages 1–18, Denver, CO, October 1988. USENIX.

[17] R.N. Taylor, F.C. Belz, L.A. Clarke, L.J. Osterweil, R.W. Selby, J.C. Wileden, A.L. Wolf, and M. Young. Foundations for the Arcadia environment architecture. In *Proceedings of SIGSOFT '88: Third Symposium on Software Development Environments*, pages 1–13, Boston, MA, November 1988. ACM SIGSOFT. Appears in *ACM SIGSOFT Notes*, Vol. 13, No. 5.

[18] J.C. Wileden, L.A. Clarke, and A.L. Wolf. A comparative evaluation of object definition techniques for large prototype systems. *ACM Transactions on Programming Languages and Systems*, 12(4):670–699, October 1990.

[19] A.L. Wolf. Abstraction mechanisms and persistence. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, September 1990.

[20] A.L. Wolf, L.A. Clarke, and J.C. Wileden. The AdaPIC Tool Set: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, SE-15(3):250–263, March 1989.

[21] M. Young, R.N. Taylor, K. Forester, and D. Brodbeck. Integrated concurrency analysis in a software development environment. In *Proceedings of the SIGSOFT '89 Third Symposium on Software Testing, Analysis, and Verification (TAV3)*, pages 200–209, Key West, FL, December 1989. ACM SIGSOFT.

[22] S.J. Zeil and E.C. Epp. Interpretation in a tool-fragment environment. In *Proceedings of the 10th International Conference on Software Engineering*, pages 241–248, Singapore, April 1988. IEEE Computer Society.

Porting and Extending the C++ Task System with the Support of Lightweight Processes

Philippe Gautron
Rank Xerox France & LITP
Université Paris VI
4 place Jussieu, 75252 Paris CEDEX 05, FRANCE
gautron@rxf.ibp.fr

Abstract

The C++ task system is a library designed to support concurrent activities programming. This paper presents an implementation of this task system based on a minimal set of C primitives, extracted from the Sun Lightweight Process library.

The scheduling of the original system is based on a single strategy: no preemption, no priority, FIFO mode. Our implementation extends the system support to LIFO mode, priorities and user-controlled scheduling.

Our first motivitation is to study the feasibility of both the porting and the extension of the library. Our second motivation is relative to the implementation of a C++ interface to a standard C library. Our experiment demonstrates that extensions can be achieved with language supports such as inheritance and the placement syntax, but that non-portable code can be necessary to interface to a C library.

1 Introduction

The C++ task system is a library designed to support concurrent activities programming. A *task force* is a collection of co-operative activities computing towards a common goal. The library was designed in 1980 [Stroustrup 80] and was part of the rationale of early versions of C++. It is part of the standard AT&T distribution [AT&T 89] and is a typical example of object-oriented development: the current interface is (almost) similar to its first design, whilst the implementation has been revised at several occasions. In particular, the current version has been extended to handle external events, typically UNIX[1] asynchronous signals [Shopiro 87].

This paper presents an implementation of the task system based on a minimal set of C primitives. The Sun[2] LightWeight Process C library (Sun LWP) [Sun 90] will be the system support for our experimentation.

A task force share a single address space, typically a UNIX process. The scheduling of the original task system is based on a single strategy: no preemption, no priority, FIFO

[1]UNIX is a trademark of Bell Laboratories.
[2]Sun is a trademark of Sun Microsystems, Inc.

mode. A FIFO strategy is appropriate for simulations in a pure co-routine style: environment closures allow different activities to run in pseudo-parallelism. However, when performed with a multi-thread support, asynchronous events may require different policies [Liskov and Shrira 87, Birrell 89]. Our implementation provides LIFO mode, priority and user-controlled scheduling as extentions to the task system interface.

Our first motivitation is to study the feasibility of both the porting of and the extension of the library. A first condition we imposed on ourselves is that any existing code could be recompiled with the new interface. Language supports, such as default arguments and the placement syntax, allow to cope with this problem. Our second motivation is relative to the implementation of a C++ interface to a standard C library. Our experiment demonstrates that non-portable code can be necessary. It also questions the suitability of object-oriented techniques, such as inheritance, to deal with concurrency.

The paper is organized as follows. Sections 2 and 3 briefly introduce the C++ task system and the Sun lightweight process library. Section 4 details the essential parts of our implementation. Section 5 describes the extensions. Section 6 is an assessment of our experiment. Section 7 presents some measures of performance. We conclude with the validation of our porting.
Implementation details are only discussed when relevant.

2 The C++ Task System

Many papers and tutorials describe either the internals [Stroustrup and Shopiro 87] or applications [Gautron 85, Johnston and Campbell 88, Gautron 89, SOR 89] of the task system. Only parts of the implementation relevant to our discussion are summarized here.

A class derived from the **task** class defines a task:

```
class task {
    ...
  public:
    task (char *name= 0, modetype= DEFAULT_MODE, int stacksize= DEFAULT_SIZE);
};

class myTask : public task {
      ...
  public:
           myTask ();
};
```

A new co-routine is created when the constructor of the derived class is called. A new executing environment (essentially a call stack and a set of registers) is then associated to each task.

The creation of a new task splits the program execution into two sequential flows of control: the activity of the creator and the activity of the newly created task. The *first* instantiation of the **task** class entails the implicit creation of another task, the **main** task: the starting environment is then attached to this task. In the current version of the task system, the first task is a single static instance of the `Interrupt_alerter` class, the external events manager.[3]

The task scheduler is defined in the `sched` class, a base class of `task`. The scheduler handles a single static list of "ready to run" tasks, the `run_chain`. Invoking the `sched::insert` protected member function on a task introduces this task into the `run_chain`. A call to the `task::resultis` public member function ends a task before invoking the scheduler to elect a new task.

3 The Sun Lightweight Process Library

Lightweight processes are threads of control within a single address space, typically a UNIX process. They are an appropriate abstraction to deal with concurrent activities and asynchonous events.

Many lightweight process or multi-thread libraries are available. We chose the Sun lightweight process library [Sun 90] because (1) it is available on standard workstations and on a RISC architecture, and (2) it provides the basic system support we need:

- reliable context switching (reliable registers save/restore),
- efficient allocation of protected stacks.

Unlike the SRC [Birrell 89], MACH [Cooper and Draves 87] and CHORUS [Armand and al. 89] threads, the Sun library is implemented in user mode: the operating system only knows the system process encapsulating the different lightweight processes.

In the rest of the paper, we will refer to the Sun library as the LWP library and the word process will designate a lightweight process.

4 Porting the Task System

Porting the task system concerns basic operations of the task management: task creation, stack allocation, context switching, task deletion. In the original task system, these facilities are implemented by both C++ routines and by assembler code. Porting the task system on the LWP library results in suppressing some routines and the assembler code, unfortunately replaced by a (smaller) new assembler code, and in modifying some other routines.

This section presents the LWP routines we rely on and explains why some assembler code is necessary.

4.1 LWP Routines

Six LWP routines compose the basic system support to the task system:

-
  ```
  lwp_create (thread_t *tid, void (*func)(), int priority, int flags,
              stkalign_t *stack, int nargs, int arg1, ... , int argn)
  ```
  creates a new process. The process identity is filled by the LWP library in the `tid` parameter. The function `func` is executed with `nargs` arguments (`arg1`, ..., `argn`) pushed onto the `stack` argument. `Priority` and `flags` can alter the state of the process.

[3] The fact that any order is imposed on the initialization of static objects is not relevant to our discussion.

- `lwp_self (thread_t *tid)`
  fills the identity of the process actually running. Our library calls this function only once, to identify the `main` task.

- `lwp_yield (thread_t tid)`
  allows the running process to yield control to the process argument.

- `lwp_destroy (thread_t tid)`
  allows to terminate the process argument.

- `lwp_setstkcache (int min, int num)`
  creates a pool of `num` stacks with at least `min` bytes each.

- `lwp_newstk ()`
  assigns a stack to a process.

The process scheduler, intrinsic to the LWP library, is *never* used, and each process will be created by the `task` constructor with the same process priority. The task management is superposed to the process management and the different LWP primitives are wittingly elected by the task scheduler.

4.2 Task Creation

Task creation is the main issue. Indeed, the task system allows to create tasks with an arbitrary number of arguments of arbitrary type. Task creation amounts to instantiating a class derived from the `task` class. A call to the `task` constructor is generated by the compiler within the body of the derived class constructor, before the user code, whatever constructor arguments are. These arguments are stored either into the stack or into registers. They must be grabbed by the `task` class constructor to be passed just as they are to the `lwp_create` routine.

The issue we have to solve is threefold:[4]

- save the caller environment into the process environment.
- set the address parameter of the process creation primitive to the first instruction following the call to the `task` constructor within the body of the derived class constructor.
- set the return address of the task creator to the next instruction following the task instantiation.

These operations require to access to registers values: assembler code is the only solution. The constructor of the `task` class concentrates the delicate points. Unlike the original task system, this constructor has been re-written in assembler code[5] and is the unique assembler routine of our library. This code is outlined in the next section.

[4]See [Shopiro 87] for exhaustive explanations.

[5]In fact, to limitate the assembler code, the `task` constructor is split into three member functions: `task::before`, assembler routine, `task::after`.

4.3 Assembler Code

This section brievely describes the assembler part of the task constructor.

Our strategy is to allocate an additional structure per task to copy the "hot" data (essentially the registers and the process entry point) from the caller environment. The trick is to create a new process with the address of an internal function (lwp_launch), with only one argument, the this pointer of the task. The data will be restored into the new process environment, as offsets of this, before the jump to the process entry point.

Simplified pseudo-assembler code looks like:

```
task::task:
  // at this point we are always under the derived class constructor stack frame
    if this == 0 then allocate task_memory;
    save registers;
    save entry_point;          // first instruction of the new process

  // push stack frame for the task constructor
    push stack_frame;          // task::task stack frame
    call sched::ctor ();       // inline initialization of base classes
    call task::before ();      // task initialization
    if first_task then return;        // main task is current process
    call lwp_create (..., lwp_launch, ..., 1, this);
    call task::after ();       // call lwp_yield
    pop stack_frame;
    return;                    // restore stack frame *twice*
                               // to return to the caller environment

lwp_launch:
    push stack_frame;
  // restore environment from this, including stack content
    restore stack;             // from caller's stack and frame pointers
    restore registers;
    jump entry_point;          // jump to the first user instruction
    assert never_reached;
```

The task scheduler takes charge of process deletion. Returning from the call to lwp_launch is a user error, typically when a call to task::resultis has been forgotten. An appropriate message is then printed and the program quietly exits.

This part of the code is obviously architecture-dependent.

4.4 Compiler-dependent Code

Three parts of any constructor code are compiler-dependent. Indeed, C++ does not impose standard object allocation, standard virtual tables management and standard name encoding.[6]

In our implementation:

- object allocation is performed if the this pointer of the task is null.[7]

[6]The two compilers we have used, AT&T cfront and g++, radically differ on these three points.

[7]We assume the "this==0" rule a general rule to allocate memory. Object allocation may occur before the constructor call (g++) or within the constructor body (AT&T cfront).

- the pointers to the virtual tables[8] are updated within the body of `task::before`, using the *placement syntax*. This technique will be detailled in section §5.2.
- name encoding is (conceptually) our *only* compiler-dependent code.

5 Extending the Task System

A noticeable exception to the FIFO strategy of the original task scheduling is the interrupt alerter. The task system manages this particular task in LIFO mode. Explicit tests are introduced in the algorithm of the scheduler to take into account this special case. A LIFO mode is not the best way to manage interruptions. A priority-based scheduling is more suitable and, in fact, the LIFO mode is just a way to simulate a high priority.

The following paragraphs present our extensions to the task system in order to introduce different scheduling policies: LIFO mode, priorities and user-control. In particular, the interrupt alerter task is re-implemented as a high priority task.

5.1 The task Class Interface Revised

Two arguments are added to the `task` constructor. Default values guarantee the compatibility with existing code. The extended signature (extra arguments will be explained below) is the following:

```
class task : public Sched {
        ...
    public:
        task (char *name = 0,
              modetype mode = DEFAULT_MODE,
              int stacksize = DEFAULT_SIZE,
              int priority = DEFAULT_PRIORITY,
              objtype otype = TASK);
};
```

5.2 The Lifo Class

Introducing LIFO mode in the task system scheduling may seem trivial. The `task::insert` member function is turned into a virtual function, and the `Lifo::insert` member function overrides it. These functions are quite similar and only one part of the original algorithm needs to be modified: the task on which the `insert` function is applied is introduced at the beginning of the `run_chain` instead of the end. The complete interface of the `Lifo` class is the following:[9]

```
class Lifo : public task {
        virtual void insert (int delay, object* alerter);
    public:
        Lifo (char *name = 0,
              modetype mode = DEFAULT_MODE,
              int stacksize = DEFAULT_SIZE,
              int priority = DEFAULT_PRIORITY,
```

[8]Multiple inheritance is used.

[9]Inlining of the `Lifo` constructor, as well as inlining of the `Process` constructor (see §5.4), is essential to insure a correct return point from the `task` constructor.

```
                   objtype otype = LIFO)
                : task (name, mode, stacksize, priority, otype) { /* empty */ }
};
```

Task creation does not involve any overhead and the task class facilities are inherited. The exact type of the current task is notified to the task constructor by the otype argument.

More complex is the management of the main task. This task is created before any user task, in FIFO mode (default). A first choice should be to let this task in this mode, but that could conflict with users' intentions. The arbitrary strategy we have adopted is to conform the main task to the mode of the *first* user task. The main task is created as a task instance and its *effective* class is changed, on the fly, if needed, into a Lifo instance. The *placement syntax* [Ellis and Stroustrup 90, p. 60] turns out to be the appropriate language support. The Lifo class does not own additional data members: all that we need is to modify the pointers to the virtual tables in order to subsequently call the Lifo::insert member function. This modification needs the collaboration of the whole system hierarchy and is purely syntactic. All the default constructors must be overloaded, and the additional declarations look like this:

```
class object {
        ...
    protected:
        enum OverloadCtor { noop };
        object (OverloadCtor) {}
};

class sched : public object {
        ...
    protected:
        sched (OverloadCtor o) : object (o) {}
};

class task : public sched {
        ...
    protected:
        task (OverloadCtor o) : sched (o) {}
};

class Lifo : public task {
        ...
    protected:
        void* operator new (size_t, task *t) { return t; }
        Lifo (OverloadCtor o) : task (o) {}
};
```

The creation of the first user task can be easily detected within the body of the task constructor. The following instructions are added to the code of the task constructor:

```
// assume at this point thistask is pointing to the main task
    if (is_first_user_declaration ()) {
        if (otype == LIFO){
            OverloadCtor o = noop;
            (void) new (thistask) Lifo (o);  // placement syntax
        }
    }
```

Data members of the main task stay unchanged except the pointers to the virtual tables, now pointing to the Lifo virtual tables. This code is performed only once and the overhead is negligeable.

5.3 Priority-based Scheduling

5.3.1 Interface Extension

Introducing priority-based scheduling in the task system is easy. Our extented library supplies 8 levels of priority. This number is a compromise between the scheduling overhead and users' needs.[10]

The single chain of tasks handled by the scheduler is replaced by a vector of 8 chains of tasks. The scheduler algorithm is standard: hight priority served first, FIFO (or LIFO) mode within a same priority.

Code modifications are relative to the `task::insert` and `Lifo::insert` member functions. A `priority` argument is added to the signatures of the `task` and `Lifo` constructors (see sections §5.1 and §5.2). The default is to create a task with a medium priority (4).[11] The `main` task priority is tuned according to the priority of the first user task.

5.3.2 Interrupt Alerter

Different strategies may be considered for the interrupt alerter management: FIFO or LIFO mode, priorities. The more exact mode has been chosen: a `Lifo` instance with the highest priority. The interrupt alerter becomes thus a standard task (the scheduler has no longer particular knowledge of this task and the corresponding code has been deleted).

5.4 User-Controlled Scheduling

Our model of user-controlled scheduling was inspired by the Smalltalk [Goldberg and Robson 83] process model. A `Process` class, derived from the `task` class, supplies the interface to the task system.

In accordance with the model, a `Process` instance is created in *suspended* mode. A running task must explicitly insert this last task in the system chain (in FIFO mode) before yielding control. A call to the scheduler is then performed; control is relinquished to the head task of the chain, not necessarily to the newly created task. Priorities are checked to elect the running task.

Many facilities was already supported by the task system and most `Process` member functions are merely renamed. The SUSPEND mode is added to the `modetype` enumeration of the original task system. Part of the `Process` class interface looks like this:

```
class Process : public task {
        ...
    protected:
        void* operator new (size_t, task *t) { return t; }
        Process (OverloadCtor o) : task (o) {}
    public:
        Process (char *name = 0,
               modetype mode = SUSPEND,
               int stacksize = DEFAULT_SIZE,
```

[10]The Smalltalk process model influenced our decision.

[11]Each task of an existing code will own this priority.

```
                int priority = DEFAULT_PRIORITY,
                objtype otype = PROCESS)
              : task (name, mode, stacksize, priority, otype) { /* empty */ }
};
```

In accordance with the Lifo strategy, the main task can be possibly changed into a Process instance. To that effect, a use of a placement syntax quite similar to the declarations of the Lifo class (see section §5.2) is added to the task constructor.

The assembler code of the lwp_launch function (see section §4.3) must be slightly revised to prevent the running task from relinquishing control to the newly created task. The following code is inserted before the jump to the entry point:

```
lwp_launch:
    ...
    if modetype == SUSPEND then
         call task::process_schedule (); // mark IDLE and schedule
    jump entry_point;
    ...
```

6 Assessment: Interface to a C Library

This section assesses our porting and discusses the use of inheritance to create tasks.

Two parts of the original library are non-portable code. They are relative to the system support, and to the task creation mechanism.

The former concerns allocation and switching of hardware resources, such as registers and stacks. The corresponding code is necessarily non-portable since these resources are machine-dependent. In our porting, these supports are encapsulated by C routines of the underlying library.

The latter is relative to the task instantiation mechanism, more precisely to argument passing to the constructors. It was our earlier belief that a non-portable code could be avoided. Our implementation fails on this point. Flexibility –users may create tasks with arbitrary numbers of arguments of arbitrary type– must be paid with non-portable code. Existing solutions, typically varargs declarations [Ellis and Stroustrup 90, p. 146], are not universal and require a full control of the sources. As other example, Smalltalk supplies different methods to deal with the name and the arguments of message passing (equivalent to procedure call). Such a facility is non-portable and does not easily fit a typed language.

More, the C++ inheritance mechanism is implemented in such a way that the base class constructor is called within the body of the derived class constructor. The stack frame of the derived class constructor is always created first, and, in any case, the task class constructor has no other alternative than to copy at least this stack frame in the newly created environment.

Note that the incompatibility between inheritance and concurrency has been already related in other contexts, such as synchronization of concurrent objects [Kafura and Lee 89].

7 Performance

This section relates different measurements of performance. The results must not be considered in their raw form but interpreted in a comparative way.

Our configuration was a Sun 4/65 –16 Mips, SunOS 4.1, single-user–. Time unit is the elapsed time per task in microseconds. The number of available tasks depends on paramaters of the operating system.

- creation (including the first context switch):

number of tasks	LWP	original library	our library
5	1000	0	1000
10	1000	500	2000
25	800	400	2400
50	700	400	2600
100	700	500	2650
250	720	440	2760
500	700	490	2920
750	700	500	2960

- context switch:

number of tasks	LWP	original library	our library
5	0	0	500
10	500	0	500
25	400	0	600
50	400	200	600
100	500	300	650
250	480	320	730
500	480	520	940
750	550	650	990

- deletion:

number of tasks	LWP	original library	our library
5	0	0	0
10	0	0	0
25	200	0	0
50	400	0	0
100	300	50	100
250	320	80	120
500	340	150	160
750	380	220	220

8 Conclusion

This paper presents both an implementation and an extension of the C++ task system. Lightweight processes are the basic system support of the tasks, and three extensions are introduced: LIFO strategy, priorities and user-controlled scheduling.

A fourth extension, preemptive scheduling, was planned. This extension requires a collaboration between the LWP library preemptive scheduler and the task scheduler. Indeed, the process scheduling can occur at any time, leaving the task management in an incoherent state (the value of the `thistask` pointer for example). This extension was partially realized. It is not impossible but painful: a different interface would be more appropriate.

Our implementation does not preclude the use of other LWP library facilities (a member function to gain access to the process identity is enough and provided). The implementation was validated with different C++ and C compilers (AT&T cfront and g++, Sun-cc and gcc), with *optimization flags*. The compiler-dependent code is limited to name encoding of a few member functions. Different applications, including the standard test suite of the original library, were successfully run after recompilation. Performances are reasonable in regard to both the LWP and the task libraries (except maybe for the task creation) –clearly, threads within a single system process cannot satisfy the requirements of real-time applications, for example–. Porting the library on other architectures supporting the LWP library could be achieved in a reasonable time but is not planned. More interesting should be to port the task system on other multi-thread libraries. The constraint is weak: that the library supplies the basic primitives we have mentionned.

Acknowledgments

Helpfull comments and corrections were provided by Marc Shapiro from INRIA and by the anonymous reviewers. Francois Koughaz wrote the C++ part of the implementation as his master degree project.

References

[Armand and al. 89] Armand, Francois Frédéric Herrmann, Jim Lipkis, and Marc Rozier. Multi-threated Processes in CHORUS/MIX. *Technical Report CSTR 89-37.3*, Chorus Systèmes, St Quentin-en-Yvelines (France), 1989.

[AT&T 89] *AT&T C++ Language System Release 2.0: Product Reference Manual*, 1989. Select Code 307–146.

[Birrell 89] Andrew D. Birrell. An Introduction to Programming with Threads. *Technical Report 35*, Digital Systems Research Center, Palo Alto, California 94301, January 1989.

[Cooper and Draves 87] Eric C. Cooper and Richard P. Draves. C threads. *???*, March 1987.

[Ellis and Stroustrup 90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Programming Language.* Number ISBN 0-201-51459-1. Addison Wesley, 1990.

[Gautron 85] Philippe Gautron. Unix et multiprocessus, C++ et multitâche : Une approche logicielle de la simulation de l'improvisation dans le jazz. *PhD thesis*, Université Paris XI-Orsay, IEF, Paris (France), October 1985. Also available as Technical Report LITP 86-16, LITP, Université Paris VI - PARIS.

[Gautron 89] Philippe Gautron. An Introduction to the C++ Task System. *The C++ Report*, 1(10), November 1989.

[Goldberg and Robson 83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Johnston and Campbell 88] Gary M. Johnston and Roy H. Campbell. A Multiprocessor Operating System Simulator. In *Proc. C++ Conference*, pages 169–181, Berkeley, CA (USA), October 1988. USENIX.

[Kafura and Lee 89] Dennis G. Kafura and Keung Hae Lee. Inheritance in Actor Based Concurrent Object Oriented Languages. In *ECOOP'89*, Nottingham (GB), July 1989.

[Liskov and Shrira 87] Barbara Liskov and Liuba Shrira. Promises: An Efficient Procedure Call Mechanism for Distributed Systems. *Programming Methodology Group Memo 60*, M.I.T.Laboratory for Computer Science, Cambridge, MA 02139, November 1987.

[Shopiro 87] Jonathan E. Shopiro. Extending the C++ Task System for Real-time Control. In *Proceedings and additional papers, C++ workshop*, Berkeley, CA (USA), November 1987. USENIX.

[SOR 89] SOR. SOS reference manual for prototype V4. *Rapport Technique 108*, INRIA, Rocquencourt, June 1989.

[Stroustrup and Shopiro 87] Bjarne Stroustrup and Jonathan E. Shopiro. A Set of C++ Classes for Co-routine Style Programming. In *Proceedings and Additional Papers, C++ Workshop*, Berkeley, CA (USA), November 1987. USENIX.

[Stroustrup 80] Bjarne Stroustrup. A Set of C Classes for Co-routine Style Programming. *Technical Report CSRT 90*, AT&T, Murray Hill NJ (USA), November 1980. Revised (1) July 1982, (2) November 1984.

[Sun 90] *Programming Utilities and Libraries: Lightweight Processes*, March 1990. Sun 825–1250–01.

Concurrent Real-Time Music in C++

David P. Anderson †‡ <anderson@snow.berkeley.edu>

Jeff Bilmes † <bilmes@icsi.berkeley.edu>

†International Computer Science Institute
1947 Center Street
Berkeley, CA 94704

‡Computer Science Division, EECS Department
University of California at Berkeley
Berkeley, CA 94720

Abstract

MOOD is a C++-based programming system for algorithmic and interactive music generation. MOOD uses multiple concurrent processes to generate different aspects of musical structure (pitches, rhythm, dynamic variation, etc.). It is composed of three layers. Layer one supplies deadline-scheduled lightweight processes and real-time event generation. Layer two allows processes to be collected into hierarchical group structures, with associated "virtual time systems" and nested musical transformations. Layer three provides pitches, scales, notes, rhythm specification, and higher-level musical abstractions. MOOD derives several benefits from C++ features such as inheritance and operator overloading: 1) a simple and versatile syntax for music representation; 2) a clean, layered structure for the internal scheduling mechanisms; 3) easy factorization of the machine-dependent parts (MOOD now runs on Sun 3 and 4 workstations under UNIX, and on the Macintosh).

1 Introduction

MOOD (Musical Object-Oriented Dialect) is a programming system for note-level computer music (e.g., computer control of MIDI [Int89] synthesizers). Unlike standard music sequencer programs that represent music as lists of note data structures, in MOOD the music is represented by the code itself; hence MOOD can specify musical algorithms as well as scores. We intend MOOD to support a variety of musical activities, including 1) algorithmic composition; 2) interactive performance environments, and 3) programmed score interpretation.

We implemented MOOD as a set of C++ [ES90] classes. As discussed by Pope [Pop89], object-oriented programming languages are useful for computer music since they provide design techniques such as composition, refinement, factorization, and abstraction. Our additional reasons for using C++ include:

1) Familiarity: Programmers familiar with C++ can immediately use MOOD. Furthermore, MOOD can benefit from C++ development activities in other areas, such as user-interface toolkits.

2) Clean syntax: The operator-overloading features of C++ allow us to provide concise and intuitive syntax for common musical structures.

3) **Speed:** Optimizing compilers are available for C++, and there is no built in garbage collection. These properties improve timing accuracy.

4) **Portability:** C++ runs on a variety of machines and operating systems, and makes it easy to encapsulate system dependencies.

5) **Extensibility:** Inheritance make it possible for programmers to extend and customize the features of MOOD for particular musical styles or applications.

MOOD is composed of three layers. Layer one supplies deadline-scheduled lightweight processes and real-time event generation. Layer two supplies hierarchical "virtual time systems" that allow the nesting of musical transformations. Layer three provides pitches, notes, scales, rhythms, and higher-level musical abstractions. Figure 1 shows the MOOD class hierarchy and its division into layers.

2 Layer One: Basics

Layer one provides the rest of MOOD with lightweight processes, accurately-timed event performance, and other low-level features. MOOD uses the scheduling model of FORMULA [AK90].

2.1 Real-Time Processes

MOOD uses several specialized types of processes. The `PROCESS` class abstracts the features common to these types. These include a stack, an SP save slot, and member functions for context switching and stack initialization. The `RT_PROCESS` class also inherits `SCHED_REQ`, adding the necessary state for real-time scheduling (see below). An `ARGS` object stores the arguments (and their number and types) to be passed to the initial procedure executed by the process. A process might be created as follows:

```
ARGS args;
PROCESS *p;
args << 5 << 3.5;
// the new process will execute foo(5, 3.5)
p = new RT_PROCESS(foo, args);
```

2.2 Process Scheduling

Real-time scheduling is encapsulated in the classes `SCHEDULER` and `SCHED_REQ`. `SCHED_REQ` abstracts the notion of "schedulable entity", with virtual member functions `run()` and `preempt()`. For example, the implementation of `run()` in `RT_PROCESS` simply switches to the process. A `SCHED_REQ` also includes `TIME` members `deadline`, `time_position`, `maxdel`, and `mindel`. `time_position` is the real time for which events (such as notes) are currently being computed by the entity. `time_position` may be greater than the current real time but if it exceeds it by more than `maxdel` (i.e. $time\_position > currentSVT + maxdel$), the entity is temporarily put to sleep. `deadline` is the entity's scheduling priority, and is equal to `time_position - mindel`; thus `mindel` can be used to prioritize processes (such as input handlers) with similar time positions.

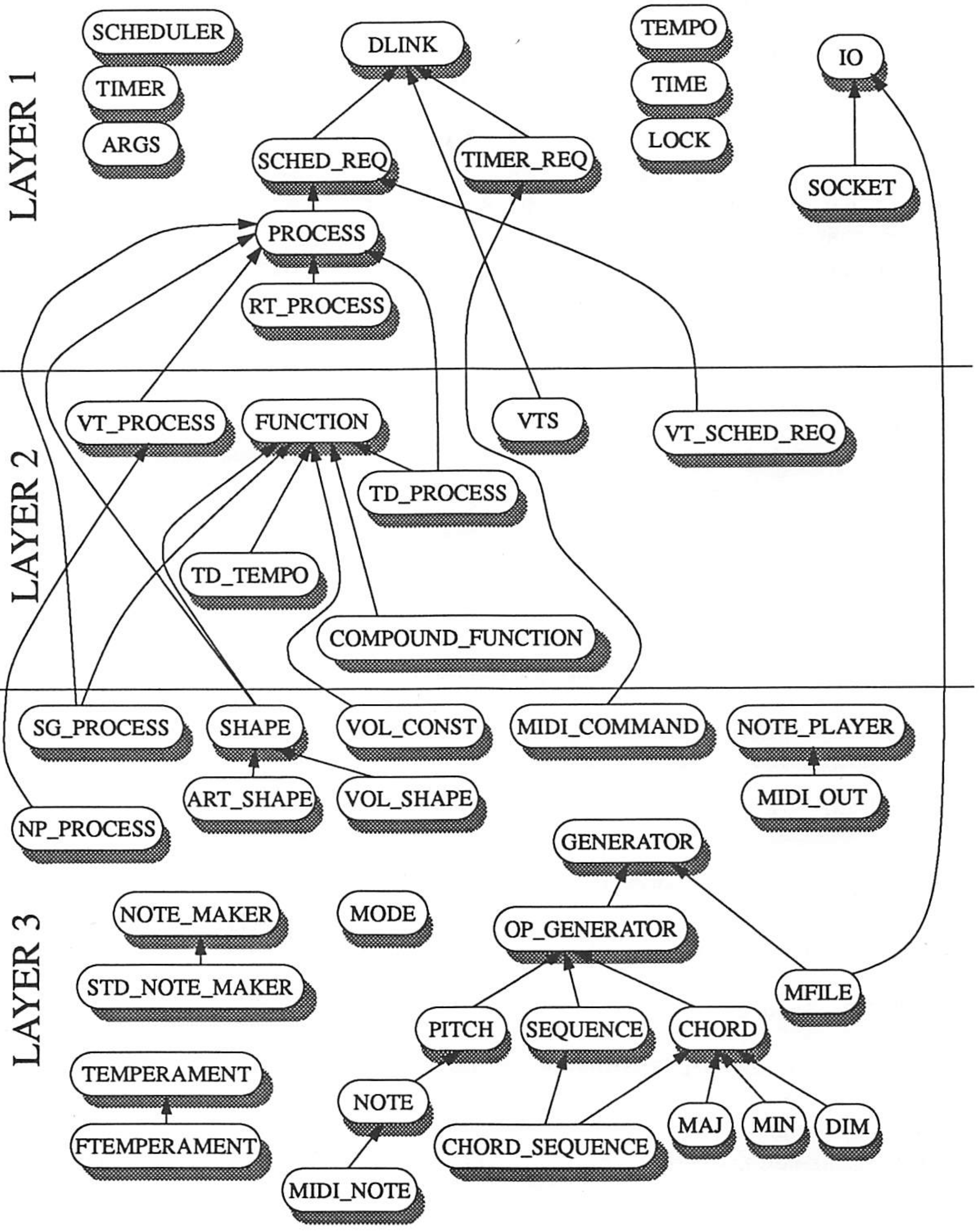

Figure 1: The MOOD class hierarchy and layers.

Earliest-deadline-first CPU scheduling is used: a SCHED_REQ object runs only when it has the earliest deadline, and runs until it changes its time position and deadline using:[1]

```
    SCHED_REQ::set_time_position(TIME);
```

After an RT_PROCESS is created and initialized, it can be made runnable using:

```
    SCHEDULER::make_runnable(SCHED_REQ*);
```

Scheduling is *preemptive*: if a process with an earlier deadline than the current process P is made runnable (by an interrupt handler or by P itself) it preempts P.

Real time is expressed in units of system virtual time (SVT). The number of units of SVT per clock period can be varied slightly to phase-lock MOOD to an external timing source. The classes TIME and TEMPO represent times (absolute or relative) and time scaling factors respectively. On the MC68020, TIME and TEMPO objects are 64 and 32 bit fixed point integers; on the SPARC, they are 64 and 32 bit floating point values. C++'s operator overloading facilities make the implementations of TIME and TEMPO transparent to users of the classes.

When possible, events are computed before they are performed (i.e., the deadline of the currently running process may exceed the current SVT). The member variable maxdel determines the maximum amount by which an RT_PROCESS may run ahead of real time, and thus limits its response latency for asynchronous I/O events. system_mindel, a TIME member of SCHEDULER, is used to keep processes from falling too far behind schedule. If the following condition holds:

$$earliest\ deadline \leq currentSVT - system\_mindel$$

then SVT is not advanced.

2.3 Event Generation

To improve timing accuracy, MOOD separates the computation of a note and its parameters (pitch, volume, etc.) from the action that causes the note to sound. This is especially useful when using MOOD with a DSP system as a synthesizer. The class TIMER encapsulates timed performance of output actions, via

```
    TIMER::insert_request(TIMER_REQ*).
```

TIMER_REQ is an abstract class; its derived classes define "action routines" and their parameters. A Process plays notes at a specific time by: 1) advancing its time position, 2) computing the notes to be performed at that time, and 3) calling TIMER to schedule the playing of the notes. Hence a process is usually computing notes that will be sounded at a later time.

TIMER uses a separate *timer process* for event performance. On each clock interrupt, TIMER checks if any events are pending and if so makes the timer process runnable. The timer process checks if SVT has reached or passed an event's time, calling the appropriate routine if it has. The accuracy of event timing is limited by the clock period.

[1]We use the scope resolution operator here to show which class contains the member function.

2.4 Asynchronous I/O

MOOD provides asynchronous I/O, i.e., a facility for a process to do a blocking read from a descriptor without causing the whole UNIX process to block. This is encapsulated in the `IO` class, whose constructor takes a file descriptor and which provides `read()` and `write()` operations, implemented using `SIGIO` and `select()`. A flag indicates whether `read()` and `write()` operations should return when 1) some I/O has been done or 2) the entire request has been done.

2.5 UNIX Implementation

In the UNIX implementation of MOOD, the timer and I/O interrupts are signals. The signal handlers may wake up processes (e.g., the timer process). If one of these processes has a deadline before the current process, preemption is needed. This is done by changing `sc_pc` in the handler's `struct sigcontext` to the address of an assembly routine `preempt`. This routine saves the complete context of the interrupted process and does a (non-preemptive) context switch to the process with the earliest deadline.

MOOD uses *virtual interrupt masking* for critical sections. This technique uses *mask level* and *request* variables. If a signal handler finds that the mask level is nonzero, it sets a bit in the request word and returns. `mask_ints()` increments the mask level. `unmask_ints()` decrements the mask level and, if it is zero and the request word is nonzero, calls routines that do the work of the clock or I/O signal handlers.

UNIX is not a real time operating system, and there are noticeable timing delays when there is other system activity. On the SPARCStation, we have adapted MOOD to use a MIDI device driver that keeps an output queue in the kernel, and performs events at the hardware interrupt level. This greatly increases the accuracy and resolution of event timing.

3 Layer Two: Virtual Time

Layer one allows processes to schedule events in real time. Layer two extends this by supporting *virtual time systems*: coordinate systems for time that can run faster or slower than real time, modeling the way a musician varies tempo during a performance. Each virtual time system is represented by an object of class `VTS`, whose state includes *inner* and *outer* time positions (generalizations of layer one's deadlines). An object of class `FUNCTION` defines the mapping (when inner time is incremented by x, the `FUNCTION` "deforms" x and the result is added to the outer time).

`FUNCTION` is an abstract class. Simple derived classes multiply x by a constant, or call a function to deform x. `TD_PROCESS` deforms x by doing a coroutine switch to a *time deformation process.* This process defines a "tempo function" by calling primitives

```
// linear tempo change
td_seg(TIME dt, TEMPO start, TEMPO end);
// pause
td_pause(TIME dt);
```

whose implementations handle the context-switching details. Finally, `COMPOUND_FUNCTION` encapsulates a list of `FUNCTION` objects, which are called in order when the `COMPOUND_FUNCTION` object is called.

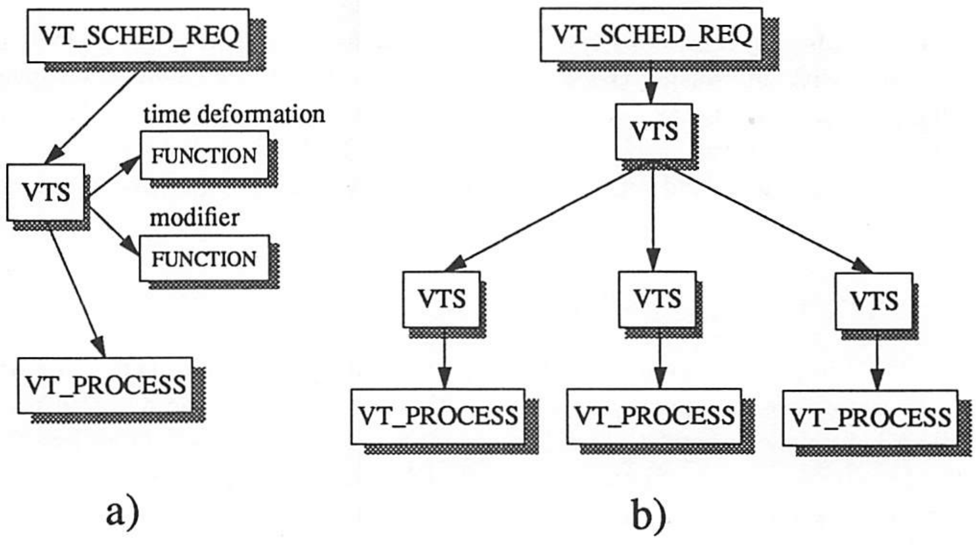

Figure 2: Examples of a single virtual-time process (a) and a group of three such processes (b). Attached to each VTS are two `FUNCTION`s: a "time deformation" that controls its tempo, and a "modifier" that is applied to any notes played by descendant processes.

Processes can be organized into a hierarchy of groups, each with their own virtual time system (see Figure 2). A top-level group is represented to the `SCHEDULER` as a `VT_SCHED_REQ` object, which inherits from `SCHED_REQ`. This `VT_SCHED_REQ` is linked to a VTS whose outer time corresponds to real time. Each VTS represents (and contains pointers to) either a group of VTSs or a single `VT_PROCESS`. A `FUNCTION` attached to a `VTS` affects all of its descendants.

A `VT_PROCESS` can schedule events using `VT_PROCESS::timer_request(TIMER_REQ);` This stores the (real-time) time position of the caller's topmost VTS ancestor in the `TIMER_REQ`, then calls `TIMER::timer_request()`. A `VT_PROCESS` can change its time position using `VT_PROCESS::time_advance(TIME dt)`. Timing is expressed in the inner time of its `VTS`. The mapping to real time is the composition of all the `FUNCTIONS` along the branch of the `VT_PROCESS`.

When a `VT_SCHED_REQ` is executed, it does a context switch to its "earliest" `VT_PROCESS` descendant, whose time position determines the time position of the `VT_SCHED_REQ` and therefore its deadline. For efficiency, VTS groups are time-ordered lists, and each VTS stores a pointer to its earliest `VT_PROCESS` descendant.

Each VTS also includes a *note modifier* `FUNCTION` used by layer three (see below). Finally, layer two allows a `VT_PROCESS` to schedule *future actions* at times other than its current time position. These are stored in a "future action queue", which is traversed by `VT_PROCESS::time_advance`. This makes it convenient to schedule key-up commands whose timing may be intermixed with future key-down commands.

4 Layer Three: Music

Layer three of MOOD provides music-specific features. The design uses abstract classes and inheritance to provide an "open framework" in which new features can be added easily.

4.1 Note Playing Processes

NP_PROCESS (note-playing process) adds musical features to VT_PROCESS. A NP_PROCESS contains a NOTE_MAKER and a NOTE_PLAYER object as instance variables. Objects subclassed from GENERATOR, representing notes or groups of notes, are sent to a NOTE_MAKER using NOTE_MAKER::operator<(GENERATOR&) for successive events and NOTE_MAKER::operator<=(GENERATOR&) for simultaneous events. The NOTE_MAKER will then ask the GENERATOR to supply it with event information by calling the virtual member function GENERATOR::apply(NOTE_MAKER&);. Subclasses of GENERATOR redefine the apply() routine to send their component objects back to the NOTE_MAKER. Using this method, subclasses of GENERATOR may produce a variety of musical structures.

A NOTE object represents a single note. It adds duration, volume, and duty (the fraction of duration during which the note sounds) to PITCH objects. A NOTE_MAKER object has specific versions of operator<() for PITCH objects, subclasses of NOTE objects, and ints. These routines can take a PITCH or a partially constructed NOTE (e.g., lacking volume) as input, and produce a fully constructed NOTE. They do this in the standard version of NOTE_MAKER by applying the note modifier FUNCTIONs of the ancestor VTSs, in order. For example, these FUNCTIONs might modify the volume of notes, supply a duration, or change the duty.

A NOTE_PLAYER object takes complete NOTES as input and handles them in a subclass-dependent way. The standard NOTE_PLAYER class plays the note via MIDI, using TIMER to schedule the output actions (usually note on and off MIDI events) and SCHEDULER to advance its time by the duration of the note. Other subclasses write the note to a file, or send it to another process.

The NP_PROCESS typically computes a sequence of PITCHs and uses its NOTE_MAKER to convert these to NOTEs which then feeds them to its NOTE_PLAYER. Rhythmic figures are obtained by note modifier functions that use a SG_PROCESS (sequence generator process) set up by the NP_PROCESS. A NP_PROCESS may access its NOTE_MAKER using the symbol NM. Assuming C, D, E, and F are PITCH objects, a succession of notes may be played as:

```
NM < C < D < E < F;
```

4.2 Pitches, Modes, and Temperament

PITCH objects, representing frequencies, maintain a pitch number which can be used as a MIDI note number or as a value to generate a frequency. PITCH objects also contain, as member variables, MODE and TEMPERAMENT objects. The pitch number of a PITCH object can change in either pitch steps (called pitch step deltas) or mode steps (called mode step deltas). When a PITCH changes by i pitch steps, the frequency the PITCH represents usually changes by i half-steps in the chromatic scale. When a PITCH object changes in mode steps, the pitch number must change by an appropriate number of pitch steps. A MODE object acts as a function with state that maps from mode step deltas to pitch step deltas. The state is simply the current position in the MODE. Therefore, when a PITCH object changes by n mode steps, it notifies its MODE object who adjusts itself by n mode positions and supplies a pitch step delta back to the PITCH object. Predefined global MODE objects include Ionian through Locrian, and, although they are not modes in the same sense, Chromatic, Major, Minor, HarmonicMinor, MelodicMinor, LydianMinor, Blues, Gypsy, Jewish, etc. The "..." mechanism is used in MODE constructors. The first argument gives the number of mode steps, and the following arguments supply the steps themselves. For example:

```
const MODE Chromatic(12,1,1,1,1,1,1,1,1,1,1,1,1);
const MODE Ionian(7,2,2,1,2,2,2,1);
const MODE HarmonicMinor(7,2,1,2,2,1,3,1);
const MODE Gypsy(7,1,3,1,2,1,3,1);
```

The TEMPERAMENT of a pitch may be used to generate micro-tonal and non-equal tempered music. When constructing PITCH objects with non-default TEMPERAMENTs, the initializer PITCH object can act as the pitch base. The pitch base is the frequency that corresponds to the first scale value the TEMPERAMENT defines. One may optionally provide an offset into the temperament that specifies where in the scale the constructed pitch should be placed. For example, if C and D are PITCH instances:

```
// p uses the C natural Major scale
PITCH p(C,PureMajor);

// q uses the D natural Major scale
PITCH q(D,PureMajor);

// r uses a Natural Major scale whose second frequency is C.
// Note: The frequency of one pitch step less than r is not the
// same as the frequency of B.
PITCH r(C,PureMajor,1);
```

Thus, the constructed PITCH object along with its TEMPERAMENT defines a mapping from integer pitch numbers to frequencies. When a PITCH is used with an FTEMPERAMENT, the FTEMPERAMENT may be used to calculate the frequency in Hertz of the PITCH. This will be useful for interfacing MOOD with a DSP system. Global FTEMPERAMENT objects are pre-defined including EqualTempered, PureMajor, PureMinor, and MeanTone. The "..." mechanism is used in FTEMPERAMENT constructors also. The first argument gives the units in divisions per octave (1200 corresponds to cents), the second argument gives the number of frequencies per cycle, and the following arguments supply the multiplicative factors for computing the successive frequencies. The following is an example of constructing FTEMPERAMENT objects.

```
const FTEMPERAMENT EqualTempered(1200,12,100,100,100,100,
    100,100,100,100,100,100,100,100);
const FTEMPERAMENT PureMajor(1200,12,70.673,133.237,111.731,
    70.673,111.731,70.673,133.237,70.673,111.731,133.237,70.673,
    111.731);
```

The default MODE for a PITCH is Chromatic, and the default TEMPERAMENT is NullTemperament (does nothing). These defaults are implemented using C++ default arguments in the PITCH constructors.

4.3 Musical Operators

Operations on musical structures applicable to a NOTE_MAKER are abstracted by the class OP_GENERATOR (operable generator). Using C++ operators, musical structures may be manipulated in novel ways.

4.4 Operations on PITCH objects

Abstracted by OP_GENERATOR, many C++ operators are defined on PITCH objects. For example, if p is a pitch, p++ increments the pitch by a mode step, p-- decreases it by a mode step, +p returns a sharpened pitch (a pitch that is one pitch step greater), -p returns a flatted pitch, p += i increases p by i mode steps, p -= i decreases it by i mode steps, p %= i increases in MODE steps but wraps at the octave, p <<= i increases p by i semitones, p >>= i decreases in semitones, p[i] is i octaves above p, and p[-i] is i octaves below. Assignment operators allow both PITCH and MODE objects to be assigned to PITCHs (assignment of a MODE to a PITCH only changes the MODE). Relational operators are defined as expected (< means less in pitch, etc.). Most binary operators are also defined as expected on pitches, preserving mathematical identities where possible. Therefore, pitch equations and expressions may be used. For example:

```
if ( p+3 >= A[i]-1 )  // i is an int
    NM < p+3 < p<<4;
else
    NM < p-2 < p>>3;
```

Global pitch constants (A through G) with default MODE and TEMPERAMENT are predefined. A[5] is 440Hz and C[5] is middle C. To reduce unnecessary bracket use, pitch constants C1 through B12 are also defined. Certain operations are not allowed on pitch constants. For example, the construct C++ is invalid since it is an attempt to modify the constant pitch C. C++ constant member functions make it easy to enforce this restriction.

With these constructs, we may write:

```
for (PITCH p = C[5]; p <= C[6]; p++)
    NM < p;
```

which will play a chromatic scale starting at middle C, and

```
PITCH p(Ionian);
for (p = C[5]; n <= C[6]; p++)
    NM < p;
```

which will play a C major scale.

4.5 SEQUENCEs, CHORDs, and MFILEs

SEQUENCE objects store an ordered sequence of OP_GENERATOR objects. A SEQUENCE will send its components to a NOTE_MAKER in the order they were loaded. Similar to a NOTE_MAKER, SEQUENCE objects are loaded with SEQUENCE::operator<(OP_GENERATOR&);. All of the operators defined by OP_GENERATOR may be applied to a SEQUENCE; the operation defined will affect all of the SEQUENCE's components. Therefore, one may easily manipulate melodies.

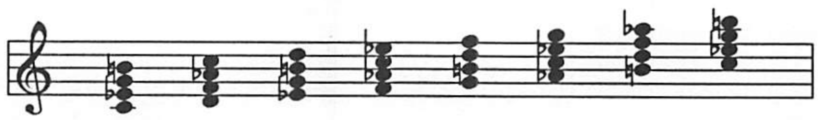

Figure 3: Chords of the C Harmonic Minor Scale

```
SEQUENCE s;
s < C4 < F4 < G4 < D4 < C5 < -B4 < F4;
NM < s;     // play the sequence
NM < s[1];  // play it an octave above.
NM < s<<1;  // play it a half-step down
```

Similar to SEQUENCE objects, CHORD objects cause its component's events to occur when applied to a NOTE_MAKER. A CHORD object's components play simultaneously however. Therefore, one may easily play the chords defined by a given mode (see figure 3).

```
CHORD ch;
ch <= C5 <= -E5 <= G5 <= B5;
ch = HarmonicMinor;
// Play the chords defined by the C Harmonic Minor scale
for (int i=0;i<8;i++)
    NM < ch++;
```

CHORD objects also define operators for changing the voicing and chord inversion. Subclasses of CHORD provide predefined chords. The root of these chords are given by a PITCH at construction time. For example, MAJ7 maj7(C4); produces a C Major 7th chord.

CHORD_SEQUENCE objects are used when both operator<() and operator<=() are needed to store a set of OP_GENERATORs. MFILE objects can be used to store sequences or chords and play them at a later time. For example:

```
MFILE mf("myFile");
if (mf.open()) {
    PITCH p = C4;
    mf.truncate();
    while (p < C5)
      mf < p++;
    mf.close();
    NM < mf; // this will reopen and play the file.
}
```

SEQUENCEs, CHORDs, and MFILEs thus make it relatively easy to manipulate melodies, chord sequences, and pitches contained in files.

4.6 Rhythm

Note durations are either associated with NOTE objects when they are constructed, or are obtained by the NOTE_MAKER (via a modifier) from a SG_PROCESS (sequence generator process). The function executed by a SG_PROCESS specifies rhythmic figures using the two macros. B(n,d) specifies n note durations of length $1/d$ (e.g., B(4,16) specifies four sixteenth notes).

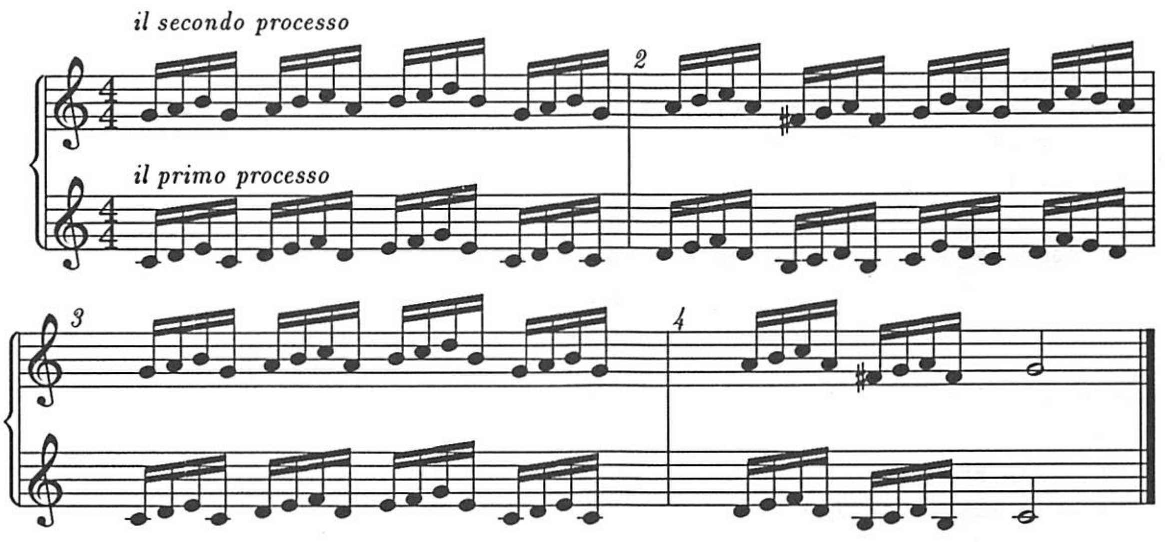

Figure 4: MOOD Score Example

F(n,d) specifies a n/d note duration (e.g., F(1,1) is a whole note duration, F(3,8) is a dotted eighth note duration, and F(7,16) is a double dotted quarter note).

5 Example

The following example demonstrates how MOOD may concisely represent musical figures. It shows how abstract musical structures may be defined (the definition of procedure `void playLine(PITCH&);`), and how concrete instances of the abstraction are made (creation of processes executing the procedure). Figure 4 shows the traditional notation.

```
// MOOD demonstration program.
#include "mood.h"

void seqGen() {
  B(56,16);  // generate 56 16th notes,
  B(1,2);    // and 1 half note
}

void playLine(PITCH& startPitch) {
  // set up an associated sequence generator
  AUX_INIT;
  SET_TSG(new SG_PROCESS((PROCEDURE)seqGen, no_args));
  SEQUENCE s1, s2;
  SEQUENCE bar1,bar2,bar4; // bar 1 and 3 are equivalent

  // load sequences with relative pitches
  s1 < startPitch < startPitch+1 < startPitch+2 < startPitch;
  s2 < startPitch < startPitch+2 < startPitch+1 < startPitch;
  // load bar 1 through 4.
  bar1 < s1++ < s1++ < s1; s1 -= 2; bar1 < s1++;
  bar2 < s1; s1 -= 2; bar2 < s1++ < s2++ < s2;
  bar4 < ++s1; s1 -= 2; bar4 < s1 < C4;

  // send sequences to our note maker
  NM < bar1 < bar2 < bar1 < bar4;
}

main() {
  NP_PROCESS *p;
  VT_SCHED_REQ *q;
  timer.start_clock();
  timer.set_tempo(2500000);
  ARGS args1, args2;

  // create the first process
  args1 < PITCH(C5,Ionian);
  p = new NP_PROCESS(std_note_maker, midi_out, (PROCEDURE) playLine, args1);
  scheduler.make_runnable(q);

  // create the second process
  args2 < PITCH(G5,Ionian);
  p = new NP_PROCESS(std_note_maker, midi_out, (PROCEDURE) playLine, args2);
  scheduler.make_runnable(q);

  scheduler.exit();
}
```

6 Current and Future Additional Work

We are currently adding facilities that will make rhythmic specification in a SG_PROCESS more flexible. DUR (duration) objects define a NOTE duration. DUR(4) is a quarter note as is DUR(1,4). REST objects are like DUR objects, except they define a rest. DUR and REST objects may be inserted into CADENCE objects. CADENCE objects are composed of DUR or other CADENCE objects and they may be used to operate on sets of rhythmic figures. Therefore, one may write:

```
// add quarter and eight note to cadence
CADENCE < DUR(4) < DUR(1,8)
```

C++ operators are defined on CADENCE objects so that rhythm can be manipulated in interesting ways. For example, CADENCE::operator<<(DUR&) and CADENCE::operator>>(DUR&) shifts the rhythmic figures defined by the CADENCE either forward or backwards in time. This is useful for musical sections that are either *behind* or *in front of* the beat. If c1 and c2 are CADENCE objects, c1 && c2 is the intersection, c1

|| c2 is the union, and c1 ^ c2 is the mutual exclusion of the rhythmic figures defined by both cadences. CADENCE::operator!() transforms all REST (DUR) objects into DUR (REST) objects of the same time length. CADENCE::operator() is used to define CADENCE objects relative to other CADENCE objects. This is useful for defining rhythmic structures commonly seen in a Frank Zappa score. Common constant DUR objects are also pre-defined. For example:

```
CADENCE c1,c2;
// load c1 with two quarter notes and a quarter note triplet.
c1 < DUR(4) < DUR(4) < DUR(6) < DUR(6) < DUR(6);
// load c2 with two half notes.
c2 < DUR(2) < DUR(4) < DUR(4);

c1 << DUR(1,128)    // shift rhythm early in cadence
c1 >> DIR(1,128)    // shift late.

SG < c1 && c2;  // intersection
SG < c1 || c2;  // union

CADENCE c3,c4;
c3 < DUR(6) < DUR(6) < DUR(6) < DUR(6) < DUR(6) < DUR(6);
c4 < DUR(5) < DUR(5) < DUR(5) < DUR(5) < DUR(5) ; // 5 per measure
SG < c1(2,4,c2);  // return CADENCE defining 5 evenly spaced rhythmic
                  // figures given by c2 in the time frame between
                  // the 2nd and 4th component of c1
```

We have plans for further MOOD development. Good synthesizer management and music output objects need to be designed so that MOOD will understand and simultaneously use different types of synthesizers and DSP systems. This will involve building subclasses of TEMPERAMENT for specific synthesizers, and subclasses of NOTE_PLAYER for different sound modules. This will enable, for example, the NeXT DSP or the SPARCstation audio output device to be used together with MIDI synthesizers.

Extensions need to be made for more sophisticated algorithmic music generation. For example, automatic harmonizer objects are planned as subclasses of OP_GENERATOR. Other subclasses of GENERATOR could be used to build musical data structures such as the TTREE [Die88].

7 Related Work

Several computer languages for music are related to MOOD. FORMULA [AK90], based on Forth, is a real-time language for algorithmic music composition. MOOD's multiple tasks and scheduling policies are derived from FORMULA. MOOD provides a cleaner and more intuitive syntax for algorithmic music specification. Also, since MOOD is written in an object oriented language, it is believed that MOOD is more easily extendible. MOOD is also integrable into other C++ user interface packages that desire musical features. MOOD, however, currently lacks an interactive environment which FORMULA uses extensively.

The Canon score language [Dan89] for computer music, written in LISP, emphasizes nesting scores and operations (or transformations) on scores. Canon provides many interesting transformations on scores that allow musical constructs to be defined and later manipulated. Like MOOD, Canon Scores are not note lists, but are themselves programs. Both MOOD and Canon can express "complex parameterized behaviors" even though Canon is written in a declarative language and MOOD in an imperative language. Users of MOOD may define abstract parameterized musical constructs either using inheritance or by writing a procedure. Canon does not have as concise a syntax as MOOD, nor does it support multiple processes or abstraction through inheritance.

The NeXT Music Kit [JB89] is an Objective-C [NeX] musical interface to the NeXT machine. It provides a large assortment of well thought out tools one needs to build sequencers and sound editors, has an interface to play notes on the NeXT DSP chip, and is object-oriented and therefore extensible. The Music Kit's syntax, however, is unwieldy (primarily due to Objective-C) and common operations on notes are not predefined as they are in MOOD. Also, the Music Kit does not provide multiple processes.

8 Conclusion

MOOD provides a powerful language base for algorithmic computer music. Unlike musical "little languages" [Lan90], MOOD provides a uniform, extensible, and familiar environment for algorithmic music composition.

The features of C++ have contributed in many ways to the MOOD design. Inheritance allows us to define a graded set of process types, and to separate scheduling policies from the entities being scheduled. It also provides a framework in different implementations of a given interface (e.g., `NOTE_MAKER`, `NOTE_PLAYER`, and `TIME` can be easily substituted). The ability to overload many operators enables us to provide a rich and concise syntax for expressing pitch structures. The availability of C++ toolkits such as Interviews [LCV87] will facilitate the integration of MOOD with a graphical user interface.

An interactive language system (like Lisp, Forth, or Self [US87]) is often useful for computer music, and most current C++ implementations are non-interactive. A second inconvenience of C++ for our purposes is that one cannot add new control structures. This precludes language features such as FORMULA's "embedded process definitions". The advantages of C++, however, outweigh these disadvantages.

MOOD currently runs on the MC68020 and SPARC lines of Sun workstations under SunOS version 3.5 through 4.1 and has been compiled both with `g++` (the GNU C++ compiler) and AT&T Cfront 2.0. MOOD has also been ported to the Macintosh under MPW C++. We plan to port MOOD to the NeXT Machine, MIPS computers, and the IBM PC and PS/2.

9 Acknowledgements

Ron Kuivila and George Homsy both contributed to the design and implementation of MOOD. Steven McCanne wrote the UNIX real-time MIDI device driver for the SPARC-Station.

References

[AK90] David P. Anderson and Ron J. Kuivila. A System for Computer Music Performance. *Transaction on Computer Systems*, 8(1):56–82, 1990.

[Dan89] Roger B. Dannenberg. The Canon Score Language. *Computer Music Journal*, 13(1), Spring 1989.

[Die88] Glendon Diener. TTrees: An Active Data Structure for Computer Music. In *Proc. International Computer Music Conference*, pages 184–188. Computer Music Association, 1988.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[Int89] The International MIDI Association, 5316 W. 57th St., Los Angeles, CA 90056. *MIDI 1.0 Detailed Specification, Document Version 4.1*, 1989.

[JB89] David Jaffe and Lee Boynton. An Overview of the Sound and Music Kits for the NeXT. *Computer Music Journal*, 13(2), Summer 1989.

[Lan90] Peter S. Langston. Little Languages for Music. *Computing Systems*, 1990.

[LCV87] Mark A. Linton, Paul R. Calder, and John M. Vlissides. Interviews: A C++ Graphical Interface Toolkit. In *USENIX C++ Workshop Proceedings*, 1987.

[NeX] NeXT, Inc., 3475 Deer Creek Road, Palo Alto, CA 94394. *Object-Oriented Programming and Objective-C.* NeXT Technical Documentation: Appendices.

[Pop89] Stephen Travis Pope. Machine Tongues XI: Object-Oriented Software Design. *Computer Music Journal*, 13(2), Summer 1989.

[US87] David Ungar and Randall B. Smith. Self: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*, 1987.

OTSO - AN OBJECT-ORIENTED APPROACH TO DISTRIBUTED COMPUTATION

Juha Koivisto, Juhani Malka
Technical Research Centre of Finland
Telecommunications Laboratory
Juha.Koivisto@tel.vtt.fi   Juhani.Malka@tel.vtt.fi

ABSTRACT

OTSO is a model and a set of software tools for developing portable communication protocols and distributed applications. Specifically, it is an environment for development, interactive testing, simulation, and execution - including distributed runtime system management. OTSO is based on the principles of object-oriented programming. For the sake of easier learning and compact implementation, OTSO is built on top of the wide-spread C++ programming language. OTSO consists of the OTSO C++ class library, and a code generator.

OTSO supports concurrency, asynchronous communication, synchronization, and transparent distribution of objects. Automatic generation of interactive user interface, support for finite state automata, and multiple concurrent service instances are also offered. OTSO is suitable for simulation and efficiency optimization, since system configuration as well as scheduling, queueing and communication methods can be customized using the OTSO class library and virtual time.

In this paper, the OTSO model, some services, and part of the implementation are described. Ways to apply C++ and OTSO for communication protocol implementation are discussed. We conclude that C++ with additional support for asynchronous communication, concurrency control and distribution seems to be very suitable for implementing communication software. Class headers are used to provide information for further code generation to implement these features.

1. Introduction

The development of communication protocol software and distributed applications sets special requirements on the programming language. Data abstraction is always important when large systems are developed. Clear separation of a service specification from its implementation, and the need to adapt to changing specifications make object-oriented techniques interesting. Portability cannot be forgotten, and performance is required in high-speed applications - that makes support for parallel processing useful. In addition, transition from the current software development system to a new one should not be too difficult. We wanted to use a widespread programming language as much as possible in order to be able to utilize the existing tools and documents. However, we could not find a language that fulfilled our requirements. But the C++ programming language [Str86] seemed to be a good starting point.

This paper describes an effort[1] to add support for distribution and concurrency to C++. We wanted to avoid making changes to C++. The basic approach was to write a C++ class library and a code generator that, using the information inherent in C++ class headers, generates C++ code to support the additional features.

The resulting model and software is called OTSO [Koi89] [2]. For the sake of easier learning and compact implementation, the sequential part of OTSO is C++ 2.0. In addition, OTSO supports concurrency, synchronization, asynchronous communication, and remote procedure calls. These and some other services are provided by automatically generated interface objects called *agents*. OTSO is aimed for development, interactive testing, execution and runtime management of the distributed system. Applications involved with persistent data and transactions are not discussed further in this paper. Instead, our special interest is in implementing communication software.

Section 2 defines the basic concepts in OTSO. The layer model of OTSO communication is described in section 3. Section 4 lists the development steps of a distributed OTSO system. OTSO services are covered briefly in section 5. Details for handling communication software specific features are described in section 6.

2. The basic concepts in OTSO

The OTSO model is built on top of the C++ programming language. Most of the features of C++ are adopted, only the optional OTSO extensions and restrictions are described further here. First we will give an idea of the basic logical concepts: objects, communication and synchronization. Then we will show how they are related to the real-world concepts, such as processes and threads.

2.1 Objects, Communication and Synchronization

An OTSO system consists of C++ objects. In order to cooperate, objects must communicate. To prevent undesirable interference in parallel execution, the control flow can be constrained by synchronization. Objects that provide services for other objects are called *service providers*, or *servers*. Objects that use the services of other objects are *service users*, or *clients*. The roles of objects can change dynamically, and most objects are both service users and providers. The *abstraction boundary* [Weg89] of an object is the set of one or more public service interfaces that it inherits and implements. In OTSO, service interfaces are defined as C++ class headers.

[1]This work is part of the Finnish Software technology programme FINSOFT which has been funded by Technology Development Centre (TEKES) and a number of Finnish companies.

[2]Otso means a bear in Finnish. OTSO also refers to Transport Service between two Objects. Earlier we used the name DVOPS.

In sequential object-oriented programming languages, there are only *passive objects*. Passive objects have a set of public, externally visible services that can be invoked like functions. It is the service users that decide what and when the passive objects do. The only thread of control is, of course, shared by a passive object and its user.

There can be many threads in OTSO. Concurrency between objects and within objects is possible. Thus, control of concurrent service requests to a shared server is needed. In sequential programming languages, explicit code must be written to take care of the shared resources: either all the clients or the server must, for example, request a semaphore before executing a method, and release the semaphore afterwards. OTSO provides an implicit mechanism where the application-specific functions need not be concerned with concurrency control. A specific object, an *agent*, is created automatically for each client-server pair to take care of the concurrent requests. The agent has two alternative approaches. In the *direct-call approach*, the agent requests access to the server, executes the method[1], and releases the server. In the *message approach*, the agent creates a message and sends it to the server. Access to the server is needed during the passing of the message. The direct-call approach can be used only when the client and the server are in the same address space (i.e. same host computer process), while the message approach can be used also with a remote server.

Whether agents or messages are used or not, the application programmer always uses the C++ function call syntax for all OTSO communication:

```
servicePointer->method(argument1, argument2, etc)
```

This can be a direct virtual function call to the non-shared server, or a call to the agent which guards a shared server. A message is needed when an asynchronous service is requested, when inter-process communication is used, or when the client and server are executed in different threads. The client and the server do not have to know if an agent and a message are used. The name of the client is passed implicitly to the server.

In OTSO, service interfaces are defined as C++ class headers and virtual function declarations. A service interface declares methods which are synchronous or asynchronous (Figure 1). The caller of a synchronous method is blocked until the method has been executed, and can be sure before continuing that passing the method has succeeded and the execution of the method has been completed (a timeout can be associated with a synchronous call). The caller of an asynchronous method does not care when the method will be executed, and cannot be sure if the execution has been completed. However, the caller can choose to wait until the service request has certainly been received by the server (passing the method has succeeded), or to continue execution without any confirmation of receipt. These alternatives are called *synchronous* and *asynchronous method passing*, respectively. Synchronous method passing makes flow control possible.

[1]We use the term *method* to refer to C++ member functions.

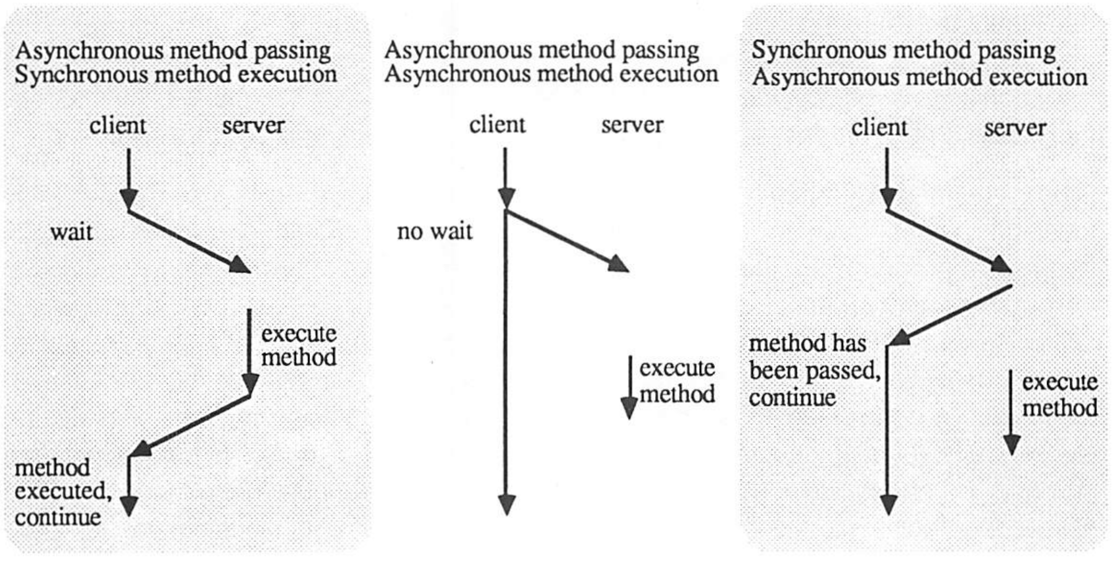
Asynchronous method passing
Synchronous method execution
client
server
wait
execute method
method executed, continue
Asynchronous method passing
Asynchronous method execution
client
server
no wait
execute method
Synchronous method passing
Asynchronous method execution
client
server
method has been passed, continue
execute method

Figure 1. Method passing and method execution.

The strong, static typing of C++ is preserved in all OTSO communication. Some requirements for the function return and argument types are discussed in section 4.1. If the function returns a value of a predefined type, the service user must wait until the answer has been returned. As an extension to C++, the return type `async` can be used. An `async` function does not return any value, and the caller can continue before the called function has been executed - synchronization between the server and the client is not necessary (asynchronous method execution is shown in Figure 1). If the return type is a class object, then the service user may need to wait. Waiting depends on the class. For instance, the `String` return type blocks the service user until the result has been returned, as in the case of predefined types. The `IntPromise` return type, on the other hand, allows the service user to continue until the returned value actually must be used - the client and server are not synchronized at the time of the call, but the returned `IntPromise` object takes care of the synchronization later[1]. In general, a *promise* is a place holder for a value that will exist in the future. It is created at the time a call is made [Lis88]. Similar principles are also called *future type message passing* [Yo86], *wait-by-necessity* [Car89], *lazy evaluation*, *Cbox*, or *concurrent call* [Buh88]. The service user only has to wait, if the answer has not arrived by the time it really is needed. An example is given in Figure 2.

Object synchronization is usually combined into communication: synchronous function calls implicitly synchronize the calling and the called object. OTSO also supports pure, explicit multi-party synchronization: objects may synchronize explicitly by naming the other synchronizing objects.

[1]Promises are not an extension to the language, they are just class objects. The waiting, however, needs some kind of wait() operation to be implemented. To avoid blocking the whole process, support for multiple threads is useful, but a helpful wait() can be implemented in a completely portable way using recursion.

```
                              //Here is an example of three different kinds of function return types.
class SP {                    //sp, sp1 and sp2 point to servers that implement the SP interface.
public:
  virtual async      as(int i);
  virtual int        bi(int i);
  virtual IntPromise pr(int i);
};

sp->as(1);                                  //Asynchronous communication,
                                            //e.g. sends a message, then continues execution.
int i = sp->bi(2);                          //Synchronous communication, waits for answer.

IntPromise answer1 = sp1->pr(1);  //Asynchronous communication with sp1
IntPromise answer2 = sp2->pr(2);  //and sp2, then continues execution.

                              //... possibly some processing ...
                              //When calculating the sum, we need the answers. If the answers
                              //have not yet been returned from sp1 and sp2, we must wait inside
                              //the implicitly called casting operator IntPromise::operator int()
                              //of answer1 and answer2.
int result = answer1 + answer2;
```

Figure 2. Function return types.

When an agent is used for concurrency control, the server is said to have a *guarded interface* to the client. If all the interfaces of a server are guarded, it is a *guarded object.* A guarded object is similar to a monitor. To support guarded interfaces, the server must be derived from the class `Guarded`. It defines a default policy for concurrency control. If the programmer wants to have finer control, the following virtual functions may be redefined:

- `request()`,
- `release()`,
- `runOrQueue(Message&)`, and
- `sendOrQueue(Message&)`

`request` and `release` correspond to the semaphore operations P and V[1]. They are needed to gain access to a shared server. If the message approach to concurrency control is used, the message is given to the server by calling the server's `runOrQueue`. A message is needed when an asynchronous service is requested, when inter-process communication is used, or when the client and server are executed in different threads. `runOrQueue` either executes the method represented by the message, or puts the message into a queue. `sendOrQueue` defines what the object does when it sends messages to other objects - by default it just sends the message.

[1]Currently `request` and `release` have no parameters, but access right parameters will be added. By redefining these functions, the programmer can allow concurrency inside the object.

The class `Runner` is derived from `Guarded`. Runner defines a virtual function `run()`. Runners usually have an *input queue* for messages. By default, `Runner::runOrQueue` puts incoming messages into the input queue, and later `run` takes messages from the queue[1] and processes them one at a time. In this way, Runners can separate the reception of a service request from serving it. Runners decide for themselves what they do when they are activated by `run`. They may also decide for themselves in which order the service requests are served (selective, conditional input). The `run` function determines when the methods are executed, separating the reception order of messages from processing order. Such separation improves reusability of sequential application methods [Car90].

To summarize, five virtual functions are needed to specify the behaviour of Runners, in addition to the application-specific functions. The behaviour of an activated Runner is defined by `run`. Concurrency is controlled by `request` and `release`. Message passing is handled by `runOrQueue` and `sendOrQueue`. These five functions are called by automatically created code, not in application specific functions Often it is enough to use the inherited default behaviour. On the other hand, redefining these virtual functions makes it possible to change the behaviour of a derived Runner class and implement the semantics of different kinds of description techniques. In one application, it often is enough to have one Runner type and use it consistently.

2.2 Distribution

The configuration of a final system (the number of processors, processes and threads, the locations of objects, etc.) implemented with OTSO is generally not known at the time of class implementation because of portability and performance optimization requirements. Thus, the application source code for the service user and provider must be independent of the configuration. *Transparent distribution* of objects must be supported. The same application source code is used for communication between objects in the same operating system process as for communication between objects on different processes, making the programs more portable. The physical configuration of the system is specified separately from the logical structure.

The unit of distribution is a `Named` object - a `Named` object cannot be split into separate address spaces. Since `Named` objects have a globally visible name, they are the *distribution boundary*: the service user and the service provider may reside in different address spaces. The class `Named` is derived from `Guarded`.

2.3 Execution

When a thread is started, it may execute some initialization code, such as C++ object constructors, and then starts a Runner that typically is a scheduler. A *scheduler* is an object that runs other Runners and decides the order in which they are run. Schedulers may schedule other schedulers, and form a scheduling hierarchy. For each thread of control

[1]The queue types, naturally, are derived from a common base class. The user of the queue only sees the common interface, `put` and `get` operations, not the implementation that defines the queuing policy such as FIFO, LIFO, or priority queue.

there is one root scheduler. A scheduler and its Runners are always in the same address space.

The `run()` function activates a Runner for an unspecified purpose, while other objects are activated by calling a specified method. When running, a Runner performs actions, then returns control to the scheduler, and waits for the next scheduling. Runners are like events that the scheduler handles. Runners may constrain the order of scheduling by priorities. The remaining *non-determinism* - the freedom to choose from several alternative scheduling algorithms - is resolved by the scheduler. There is a default scheduling algorithm, but it is hardly optimal for all applications. To find a good solution for a user's particular problem, simulation and optimization are needed. Explicit scheduling helps in this optimization.

3. The layer model of OTSO communication

Here we explain how the OTSO services for inter-object communication are implemented. The communication can be described with a layered model (Figure 3). The communication model is not concerned with execution aspects such as concurrency control and scheduling.

Each layer provides a set of services, defined in the layer's upper interface, by its internal activity and possibly by using the services of lower layers. The figure shows the path of a service request from the client to the server. The lower layer provides syntactically the same service as the peer object on the provider side of the same layer. An application programmer sees only the application layer objects - lower layers, if needed, are standard OTSO class library objects or generated by the OTSO code generator. For the service interface class `ApplSP`, the code generator generates the class `ApplSP_agent` and a set of message classes, one per method. Object instances of these classes are created automatically when needed.

The objects in the highest layer, the application layer, are application specific objects. Also some high level services of OTSO are implemented as application layer objects. The application programmer writes the `ApplSP` part of the server object, while the `Multi` part and `runOrQueue` function are inherited. Type checking at compile time ensures that the server implements the service interface `ApplSP` that the client uses. A normal C++ member function call takes place in the application layer.

The multi server layer provides several OTSO services, such as multiple concurrent server instances (with the same name), multicast, communication priorities, flow control, and alternative qualities of communication between application objects.

The message layer part of the agent converts the representation of a service request from a C++ member function call to a message object and, on the service provider side, back from the message to a function call. The `ApplSP_agent` on the client side provides the same `ApplSP` interface as the server object: class `ApplSP_agent` is derived from `ApplSP` and `Agent`. The application method implementations in the agent correspond to remote procedure call (RPC) client stub code. The `runOrQueue` function of the server object receives messages from local agents or from the lower layer. Messages are objects that know for themselves how to invoke the requested function call (server stub).

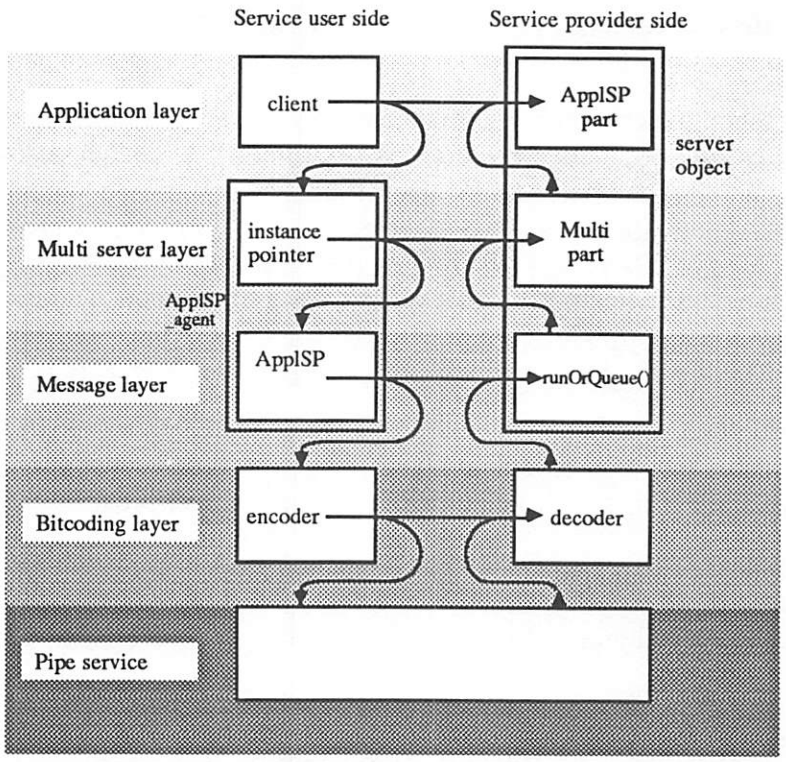

Figure 3. The layer model of OTSO communication.

The bitcoding layer translates a message to a bit stream and, on the receiving side, the bits to a message. Encoders implement the conventional C++ encoding `operator<<`, and decoders implement `operator>>`. Proper coding rules take care of the conversions that are needed in heterogeneous systems where the representations of data may vary.

The pipe service creates a communication channel and a connection between the communicating operating system processes so that bit streams can be sent to both directions. The channel preserves the order of bits. Standard methods for the inter-process communication are used to transmit bits from one address space to another through a communication network.

4. System development steps

Many approaches to concurrent computation provide extensions to a sequential language as distinct facilities and notations, for example processes, monitors and messages. However, new concepts should not make the reuse of sequential code difficult. Besides, making restrictive implementation decisions at an early phase should be avoided. Altering the system internal implementation, without changing the service provided, should be allowed to be as flexible as possible to facilitate performance optimization on a given architecture. For example, the physical locations of objects, scheduling algorithms, and input queue types can usually be changed without violating the conformance to specifications. This flexibility is achieved in OTSO by supporting transparencies: distribution transparency, transparency between message passing and direct function calls, etc.

The development of a distributed system with OTSO can be divided into four steps:

1) User specifies service interfaces	(interface class headers)
2) User writes the implementations	(implementation class headers and code)
3) User builds the logical system	(defines objects and client-server relationships)
4) User chooses the configuration	(address spaces, threads, initial locations of objects)

Figure 4. System development steps.

Service interfaces are independent of their implementations, object implementations are independent of the system architecture, and the logical system structure is independent of the final, concrete system configuration. Each step may be iteratively rewritten without repeating the previous steps. Different implementations can be written without changing the interfaces, many different systems can be put together from the same object implementations, and several configurations can be compared without touching the interfaces, objects or the logical system specification.

4.1 Service interfaces

We want to make a clear difference between the two roles of a class: the interface specification and the implementation. The interface defined by a class, sometimes called the *type*, specifies the public operations. An implementation defines one realization of a set of interfaces: the internal representation of objects and the code implementing the methods. We use the term *interface class* when referring to classes that define an interface, and the term *implementation class* when referring to classes that define an implementation. We recommend making a difference between these concepts when clear interfaces are needed. In OTSO, interface classes contain nothing but public pure virtual functions: they define the call syntax but no implementation. Interface classes are abstract base classes: no objects can be created, they are only used for derivation. If the OTSO features are needed, the application service must be defined as an interface class. Otherwise nonrestricted C++ can be used.

All C++ function declarations are not readily suitable for defining OTSO interfaces. Due to the transparency requirement, the function call semantics must be identical for local and remote communication. Therefore certain operations must be defined for the function return and argument types: operations for encoding the arguments into bit strings, and for decoding them from bit strings[1]. The OTSO class library implements these operations for predefined C++ types and for some general purpose OTSO classes. Pointers and

[1]For testing purposes, functions for printing the types in readable format, and for reading them from a test user, must also be defined.

references to `const` types are allowed; the constants are duplicated in a distributed case. On the other hand, pointers and references to non-const types are forbidden. Instead, smart pointer objects can be used. Smart pointers along with agents are generated for all interface classes by the OTSO code generator to implement safe sharing and distribution of objects.

The return value is the only value passed back to the caller, but the return type can be arbitrarily complex (as long as the operations mentioned above are defined for it). This may seem to be restrictive from the point of view of traditional coding style, but it actually is suitable for object-oriented programming and concurrent systems. Eiffel [Mey88] does not allow direct modification of formal arguments. We also require that function arguments are given a name. Ellipses (`...`) and overloading are currently not supported. The comment texts associated with the class and its operations are used as online help during interactive testing, and in textual service documents generated by the OTSO code generator.

4.2 Object implementation

An OTSO implementation class is derived from one or more interface classes, and often also from `Runner`. An implementation class defines objects that are actually created to provide the services specified in interface classes. Most of the source code is written in this step.

OTSO implementation classes are scanned by the OTSO code generator. Based on the class headers, OTSO generates code that implements a textual user interface. A test user can give service requests with desired parameters interactively to those interfaces that are defined as interface classes. In addition, OTSO can print incoming and outgoing messages with parameter values, and the data members of the running object. This allows a programming environment independent method of testing as well as the generation of a log file. These features are useful not only in the testing phase, but also in the management of a running distributed system.

4.3 The logical system: define objects and their relationships

The logical structure defines the (initial) objects in the system. Their constructor parameters typically determine if an object prefers to receive messages even though direct function calls could be used. This step also establishes the client-server relationships by connecting the pointers of clients to the servers. These relationships can be dynamically changed.

4.4 The physical configuration

The OTSO concrete system configuration defines threads, address spaces, the placement of objects into them, and the communication channels between the address spaces. This binding of logical concepts to the physical concepts is delayed until the very end to enhance portability and flexible comparison of different configurations. The method for describing the physical configuration is currently implemented for static systems. Adaptation to dynamic reconfiguration is in progress. This step should be the only difference between a system running in a development environment and the system running in the final environment. The previous steps support a set of configuration alternatives, and one of them is chosen in this step. C++ without extra precautions supports single-process systems (one address space and one thread). OTSO extends the set of possible configurations: the client and server may be placed in different address spaces.

The implementations of shared objects are classified by their tolerance of concurrency. Unguarded objects do not tolerate any concurrency and must be placed in an address space that has only one thread. Guarded objects tolerate parallel threads (or threads that have no control over the way they are scheduled) and can be placed freely. When the thread scheduling can be controlled, the implementation of concurrency control is simpler, which makes synchronous function calls more efficient.

5. Other OTSO services

In this section we briefly explain OTSO tools for *management* and *simulation* which are under development. Management is involved with the runtime control of configuration and performance. For example, it may be necessary to expand the distributed system with new parts at runtime without interfering the running system. Administrators may, for instance, add new objects to provide new services or to improve the performance. In addition to creating and deleting, it must be possible to move objects at runtime from one address space to another. Dynamic load and memory balancing are possible in systems that support object *migration*. Determining optimal object placement in a distributed system, however, is complex. The primary approach to runtime management in OTSO is not automatic but administrator control. Management may also be involved with fault detection and recovery, security, and accounting.

OTSO can also be used for discrete event simulation. Special attention has been paid to finding a good configuration under simulated load and communication delays. Processor frequencies and architectures, transmission channels and delays, as well as scheduling and queuing architectures can be compared by virtual time. Explicit scheduling hierarchies (for example a simulated processor - thread - application object) are interesting when scheduling algorithms, for example the effects of priorities and slice lengths on response times and throughput, are studied.

6. OTSO for protocol programmers

The older toolset in our laboratory for communication protocol implementation [Har86, Kar86] has been used to implement a number of both lower and upper layer OSI protocols. It is integrated into an ASN.1 compiler [ASN.1, CASN].

Based on the experiences with OTSO, we think that C++ is well suited for implementing communication protocols when extended by support for asynchronous communication and concurrency.

6.1 Service interfaces between layers

We assume that the reader has some familiarity with the ISO OSI model [OSI]. Even though we use here the OSI terminology, the discussion applies to other layered architectures as well. A basic principle in layered architectures is that changes in one layer do not affect other layers as long as the service interfaces remain the same. In addition, an entity in layer N only sees its upper and lower service interfaces N and N-1.

A layered architecture can be directly mapped to OTSO. A service interface is simply described with two interface class headers, one for upward and another for downward service primitives. The primitives are represented by pure virtual functions. A layer entity is an implementation class that inherits the class of downward primitives of the upper service interface, and the class of upward primitives of the lower interface[1]. The inherited classes contain those service primitives that can be received by the entity. They must be implemented by the protocol programmer.

In protocols and layer interfaces, communication can usually be asynchronous. Asynchronous communication gives better potential performance, because it allows more execution orders and supports parallel execution of the client and server. In OTSO, functions may have the return type `async`. These functions do not return any value, and the caller can continue before the called function has been executed. Asynchronous functions may be implemented by for example message passing. If some other data description language than C++ is used for specifying interfaces, a compiler for that language must generate interface classes. For example, an ASN.1 compiler is being integrated into OTSO [Rei90]. It generates interface class headers and the data type coding functions from ASN.1 definitions.

When layer entities are the basic modules in a protocol system implementation, the communication between neighbouring layer entities has similarities to the communication between two application processes in different computers. However, this doesn't mean that the entities have to use a seven layer model, and recursively. In section 3 we described communication between OTSO application objects with a layer model, where lower layers are used only if their services are needed.

When OSI-related protocol systems are implemented with OTSO, entities of all OSI layers are OTSO application objects. In the same sense, the N-service interface of OSI specifies an OTSO application protocol between those OTSO objects that implement N- and N+1-entities. Lower OTSO layers support flow control between N- and N+1-objects, multiplexing and handling of parallel connections, routing, asynchronous communication, quality of service, presentation syntax, etc. Most of these functions are essential even if the objects reside in the same address space.

6.2 Layer entities

Layer entities are implemented as Runners which inherit the upper and lower service interfaces. It depends on the runtime environment if real parallelism is used or not. Besides normal C++, OTSO provides an alternative way to write the method implementations: a mechanism to transform the logic and actions of textual extended finite

[1]A layer class must inherit (at least) two interfaces. With single inheritance, interface classes would have to be derived from other interfaces. This would make layer classes dependent on the implementation of other layers. Because of multiple inheritance, an N-entity implementation is affected only by changes in layer N or its interfaces.

state automata (EFSAs, state machines) into C++ class member functions. The resulting generated functions together with other user written C++ member functions define the behaviour of a class. Reducing state machines to C++ member functions is straightforward. All the state automaton triplets (state, input, output-actions) with a common input are collected into a single function that is called when that input is handled. Each function consists of a switch statement which selects the action part according to the current value of the state variable.

Scheduling algorithms can have a significant effect on the performance of any application. When choosing the algorithm, it is important to understand the dynamic behaviour of the application, in this case the protocols and their relationships. OTSO provides some scheduler classes to choose from, but such issues as operating system level scheduling and where to execute especially the code of lower layer protocols, should be considered carefully. The support for concurrency control and distribution in OTSO should make parallel processing useful at least in high speed networks and multimedia applications.

6.3 Classes for protocol programmers

Some classes for protocol implementation are included in OTSO's class library: frames, timers, OSI addresses etc. In this paper, we only give an example of a part of the class `Frame`. `Frame` is used to pass large blocks of binary coded data from one layer to another in asynchronous messages. Pass-by-reference is used to avoid copying. To avoid having multiple pointers to one block, the sender's pointer to the data block should be broken. A Frame constructor with an unusual side-effect does this automatically:

```
class Frame {
private:
  FRAME* bp;
public:
  Frame(Frame& i)
    {bp = i.bp; i.bp = 0;}
    //sender's pointer to FRAME broken!
  Frame(char*);
  ...
};
```

```
//USAGE:

async f(Frame f);
Frame myBlock = "ABC";
    // f gets "ABC", myBlock loses it
f(myBlock);
```

6.4 Abbreviations

Abbreviations are purely "syntactic sugar" which makes the application code more compact, more readable when used carefully, and easier to maintain by avoiding redundance. They are inspired by our experience of protocol software implementation, and have proved useful with applications using a large amount of communication - big messages with many parameters. For example, if the argument lists in the function declarations and calls are long, the code easily gets hard to read. When the vast majority of the arguments are passed through the function unaffected to other calls, it may be clarifying not to write the argument lists but to use an abbreviation. In OTSO, the notation `f(-)` can be used instead of the function declaration `f(int i, long j,..., S s)` or the function call `f(i,j,...,s)`. The OTSO code generator simply translates `f(-)` into `f_in`, when a declaration is expected, and into `f_out` otherwise. `f_in` and `f_out` are macros generated by OTSO from class headers. They use the same argument names (i,j,...,s) as the function declaration in the interface class definition. Thus, these names must be present in the scope where the function is called.

When C++ virtual functions are redefined in a derived class, the functions must be redeclared in the header of the derived class. In the case of large interfaces, this gets clumsy. Instead, another macro generated by the OTSO code generator for all interface classes can be used. For example, the macro `SP_interface` can be used for declaring all functions of class SP.

The abbreviations can make maintenance easier, especially in many communication protocols where the data units are defined in ASN.1. A modification in the ASN.1 specification requires updating user written code only when something else than encoding, decoding or sending to another object is done with the data unit parameters.

7. Experiences and Conclusions

The object-oriented programming paradigm has proved suitable for the implementation of communication protocol software. Clear, explicit separation of service interface from implementation is valuable. Abstract base classes and multiple inheritance facilitate reuse, since the inheritance hierarchy need not be bound to the layer system structure.

C++ seems to be a good choice. C++ class headers together with function argument names provide enough information for further code generation. On the other hand, using multiple inheritance stresses the need for parameterized types. C++ is complex and makes some parts of OTSO more complex than a pure object-oriented language would make. In fact, the philosophy behind OTSO is similar to that of C++, not insisting on the use of some paradigm, but rather offering a variety of implementation choices.

OTSO supports portable implementation of communication protocols and distributed systems. It uses C++ as the description language, and generates code to implement concurrency and distribution. Currently we have a prototype implementation of the OTSO class library and code generator. OTSO supports both distributed and shared memory systems, but most of our experience comes from concurrency in distributed architectures with message communication. After a few OSI protocols and small applications, our first impressions of using C++ and OTSO are positive. Concurrency and communication have been manageable. The implicit synchronization makes it possible to reuse simple sequential implementations for the safe implementation of shared objects. The application programmer always uses function calls, even though a function call, message passing, synchronization of a shared resource, or inter-process communication may take place.

Various communication methods are needed. Some applications use exclusively asynchronous message passing, since asynchronism is a way to avoid excessive synchronization between communicating objects and allows more concurrency. On the other hand, lacking language support for synchronous communication would make certain kinds of applications difficult to understand. Promise objects provide an implicit, data-driven synchronization.

Acknowledgments

We wish to thank James Reilly for the valuable work with OTSO, and for the comments on this paper.

References

[ASN.1] ISO, Information processing systems - Open Systems Interconnection - Specification of Abstract Syntax Notation One (ASN.1), ISO/IS 8824: 1987 (E) with DAD1.

[Buh88] P.A.Buhr, G.Ditchfield, C.R.Zarnke, "Adding Concurrency to a Statically Type-Safe Object-Oriented Programming Language", Proceedings of the ACM Sigplan Workshop on Object-Based Concurrent Programming, San Diego, 1988.

[Car89] D.Caromel, "Service, Asynchrony, and Wait-By-Necessity", Journal of Object-Oriented Programming, 2(4), p. 12-22, November/December 1989.

[Car90] D.Caromel, "Concurrency And Reusability: From Sequential To Parallel", Journal of Object-Oriented Programming, p. 34-42, September/October 1990.

[CASN] How to use CASN Compiler for implementation of a virtual task in CVOPS, vers. 1.0, Technical Report, Nokia Research Center, 1990.

[Geh] N.H.Gehani, W.D.Roome, "Concurrent C - a language for programming multiprocessor systems", Byte, December 1990, p. 327-334.

[Har86] J.Harju, A.Karila, J.Kuittinen, J.Malka: "CVOPS, a tool for the implementation and testing of computer communications software", Technical Report, Technical Research Center of Finland, Telecommunications laboratory, 1986.

[Kar86] A.Karila, "VOPS - a Portable Protocol Development and Implementation Environment", Proc. of IFIP - International Conference on Data Communications, Theory and Practise, Ronneby Brunn, Sweden, 1986.

[Koi89] J.Koivisto, J.Malka: "Introduction to DVOPS - an environment for developing distributed applications". Technical Research Centre of Finland, Telecommunications Laboratory, 1989.

[Lis88] B.Liskov, L.Shrira: "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems", Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, June 22-24, 1988.

[Mey88] B.Meyer, Object-oriented Software Construction. Prentice Hall, 1988.

[OSI] ISO, Information processing systems - Open Systems Interconnection - Basic Reference Model, ISO/IS 7498, 1984.

[Rei90] J.Reilly, "Notes for Developing ASN.1 Protocols, Using The DVOPS Development System", Technical Report, Technical Research Centre of Finland, Telecommunications Laboratory, 1990.

[Str86] B.Stroustrup, The C++ Programming Language. Addison-Wesley, 1986.

[Weg89] P.Wegner, "Concepts and Paradigms of Object-Oriented Programming", Expansion of Oct 4 OOPSLA-89 Keynote Talk", OOPSLA Conference Proceedings, 1989.

[Yo86] A.Yonezawa, J-P.Briot and E.Shibayama, Object-Oriented Concurrent Programming in ABCL/1, OOPSLA'86, Special Issue of SIGPLAN Notices, Vol 21, No 11, p.258-268, November 1986.

Availability

The terms of getting OTSO will be decided in spring 1991. It is likely that universities will get a version of OTSO free of charge for educational and research purposes by request. For more information about OTSO and the terms, contact

Juhani Malka
Technical Research Centre of Finland
Telecommunications Laboratory
Otakaari 7 B, SF-02150 Espoo, Finland
+358 0 456 5623
Fax: +358 0 455 0115
E-mail: Juhani.Malka@tel.vtt.fi

Checked Out And Long Overdue: Experiences in the Design of a C++ Class Library

Mary Fontana

Texas Instruments Incorporated
Computer Science Center
Dallas, Texas, 75265

Martin Neath

Texas Instruments Incorporated
Information Technology Group
Austin, Texas, 78759

ABSTRACT

The Texas Instruments C++ Object-Oriented Library is a portable collection of classes, templates and macros for use by C++ programmers writing complex applications. Developed over a two year period, it has been used on several internal projects and undergone significant design changes and improvements. In this paper, we discuss the initial goals of the project, the design and implementation approaches considered, and some of the reasons behind our decisions. Finally, we analyze what was learned in building this library, examine the overall issue of code reuse through C++ class libraries, and suggest some guidelines that can lead to wider acceptance and use of future class libraries.

1. Introduction

The Texas Instruments (TI) C++ Object-Oriented Library (COOL) is a portable collection of classes, templates, and macros for use by C++ programmers writing complex applications. It raises the level of abstraction to allow the programmer to concentrate on the problem domain, not on implementing basic data structures, macros, and classes. In addition, COOL provides a system independent software platform to ease the porting of applications which are built on top of it. In this paper we discuss the rationale behind some of the important aspects of COOL, such as its use of polymorphism, parameterized templates, and a resumptive exception handling mechanism. We also share what we learned in designing and implementing COOL and the feedback obtained from application programmers who have used it.

Our motivation for use of C++ and development of a rich class library has its roots in our extensive experience with Lisp Machine environments. TI has had considerable success using Lisp to design and implement complex, symbolic applications, such as diagnostic expert systems and production scheduling advisors. While most customers were willing to see Lisp used for prototyping, many showed considerable resistance to Lisp Machines as delivery vehicles.

The authors may be reached via electronic mail at fontana@csc.ti.com and martin@tivoli.com.
Martin Neath now works for TIVOLI Systems, Inc., Austin, TX.

As a result, we began investigating other, more mainstream languages that can provide some of the expressiveness of Lisp and which are well supported on a variety of conventional platforms. After evaluating several languages, we decided (for mainly non-technical reasons) on C++ and began the design of a comprehensive class library.

Over the course of about one year we designed and implemented many generalized classes. We began with the basics (such as, **String**, **Date_Time**, and **Complex**) to gain experience with the language, examine possible design approaches, and understand portability and efficiency issues. We next added an implementation of Stroustrup's templates [12] (such that there would be minimal source code conversion necessary when this feature is finally implemented in the C++ language) and proceeded to design and implement a variety of parameterized, polymorphic container classes (such as, **Vector**<*Type*> and **N_Tree**<*Node*,*Type*,*nchild*>). As the project proceeded, we realized the need for an object-oriented exception handling mechanism. Since no such facility had yet been proposed, we designed and implemented a resumptive capability for raising and handling exceptions similar in spirit to the Common Lisp Condition System [2]. Finally, a comprehensive, automatic runtime type query system was completed to round out the symbolic capabilities of the library.

COOL has been an ever-changing and growing C++ class library, with considerable effort spent reimplementing internal details, adding new features, extracting common functionality into base classes, etc. As such, some constraints were necessary in order to achieve compatible and seamless integration of new or modified features. Overall, the design and development of a C++ class library has been a very valuable experience, as much for the things we learned *not* to do, as well as for the positive feedback we received for the things we did correctly. This class library is currently in use on several internal projects and is largely in a maintenance-only mode of development. We expect to make the necessary changes to support the standard parameterized template and exception handling mechanisms when those features become available in commercial compilers. For more detailed information and examples of the COOL classes, the reader is referred to the appropriate sections of the reference document, *The COOL User's Guide* [13].

2. Core Technology Components

The fundamental cornerstones of COOL are an implementation of parameterized templates, a resumptive exception handling mechanism, an automated runtime type checking facility, and consistent polymorphic operations. This functionality is implemented through an enhanced preprocessor with a sophisticated macro facility [4] which generates conventional C++ source code acceptable to any conforming C++ translator or compiler [3]. The use of this compiler independent front-end allowed us to define powerful and portable extensions to the C++ language in an unobtrusive manner. This enabled us to experiment and gain experience with a variety of proposed language extensions long before they were available in a commercial product.

2.1. Parameterized Templates

We quickly found that the development and successful deployment of application libraries such as COOL required the planned (but not yet available) C++ language feature called type parameterization. This allows a class to be defined without specifying the specific data types needed. The application programmer using the class specifies the data types for each unique use of it in the application code.

An important and useful variety of parameterized template is known as a container class. This is a special kind of parameterized class where objects of some type are structured and stored together. COOL supplies the many common containers needed by typical complex applications. These have turned out to be the most important and most used classes in the COOL library. Indeed, the container classes come closest of all the classes in COOL to fulfilling the promise of true code reuse.

Each COOL container class supports the notion of a built-in iterator that maintains a current position within the collection of objects. A set of consistently named member functions allows a program to move through the collection of objects in a sequential order and manipulate the element at the current position. This might be used, for example, in a function that takes a pointer to a generic container object. The function can iterate through the elements in the container by using the current position member functions without needing to know whether the object is a vector, a list, or a queue. The capability to easily replace one type of container with another is critical since complex applications often must change their data storage mechanisms to meet requirements that evolve over time.

Several interesting issues arose for both the COOL and application programmers in designing and using parameterized classes. First, should a template make assumptions about or enforce a specific type modifier over which the class is to be parameterized or should that be part of the usage specification? Second, how much code for a template class can be moved into a base class to reduce code replication? Third, can the source code be effectively packaged to provide a rich set of member functions without burdening applications that do not use them all? Finally, what is the most effective mechanism for introducing a new parameterized type to the compiler and arranging for the inclusion of the code for that type exactly once across file boundaries?

The answers to some of these questions seemed obvious, while others required several attempts before a comfortable and correct direction was selected. When we began considering the *Type* parameter to a template, it seemed appropriate to allow the user to control the type modifier specification. This would allow one user to use a **Vector**<*Type*> template to contain "integers", while another might select "pointers to integers". Although a single template class can satisfy both uses, some slight performance degradation and loss of efficiency may result when copying and accessing a contained object. For example, the **operator[]** may return a reference that, if it is a pointer, results in an extra pointer dereference. The design decision to not enforce a particular type modifier resulted in the copy semantics of the contained object being left to the user and the template member functions using the object's **operator=** to copy and move objects.

One early decision was to design each parameterized class to inherit from an appropriate base class that results in shared type-independent code. This reduces code replication when a particular type of container is parameterized several times for different objects in a single application. The base classes typically included class-specific data members, member functions which manipulated the current position in container classes, and member functions which raised exceptions. There would have been more code in the base classes if we had decided differently on the previous issue and restricted the type parameter to data type pointers only.

The third issue (which would not even be an issue if more sophisticated linkers were available on standard operating systems) concerns the "full-featured" versus "lean-and-mean" philosophies. After considerable analysis and experimentation, we decided to embrace both philosophies by providing rich functionality with a template fracturing capability. This mechanism splits the source file on template boundaries so that each member function is copied into and compiled from its own file. The resulting object files, one for each function, are then placed in an application archive library for use at link time. This provides for only those member functions that are actually used in an application to be pulled into the executable image.

To control the introduction of a new parameterized type to the compiler and to automate the generation of the member functions, our first attempt used **DECLARE** and **IMPLEMENT** macros that were carefully located in the application source code. This was later changed to allow a command line interface through a compilation control program, where the user specifies the parameterized type on the command line as suggested by Stroustrup with the -X compiler option [12]. This mechanism is quite usable in traditional separate compilation systems, but more elegant solutions (which might have additional benefits) are possible for use in emerging integrated C++ programming environments.

We have found through our experience with COOL that parameterized container classes are the most important part of a general C++ class library. In addition, the basic design approach taken for container classes and the way in which the open issues with parameterization are solved determine the ultimate acceptance and use of the class library by application programmers.

2.2. Exception Handling

In COOL, program anomalies are known as exceptions. An exception can be a program error such as an argument out of range, or an encapsulation of a more fundamental problem such as arithmetic overflow. We believe that the current lack of an exception mechanism in the language seriously impedes the development of flexible and portable object-oriented libraries. An exception handling system offers a solution by providing a mechanism to manage such anomalies, simplify program code, and ease portability of an application. As an interim measure, we developed the COOL exception handling scheme, which is a raise, handle, and proceed mechanism similar to the Common Lisp Condition System [2].

The COOL exception handling facility [5] provides an exception class (**Exception**), an exception handler class (**Excp_Handler**), a set of predefined exception subclasses (**Warning**, **Error**, **Fatal**, **System_Error**, and **System_Signal**), and a set of predefined exception handler functions. In addition, an easy-to-use macro interface (**EXCEPTION**, **RAISE**, **STOP**, **VERIFY**, **DO_WITH_HANDLER**, and **HANDLER_CASE**) allows a programmer to create and raise an exception, and establish exception handlers at any point in a program. When a COOL class encounters an anomaly that is often (but not necessarily) an error, it represents the anomaly in an object called an exception and then announces the anomaly by raising the exception. The application program using COOL classes has the option of providing solutions to the anomaly by defining exception handler functions and establishing exception handler objects.

When an exception handler object is created, it is placed at the top of a global exception handler stack. This stack is maintained in a similar way to that described by Miller [11]. When an exception is raised, a search for an appropriate handler starts at the top of the exception handler stack. When a match against an exception type is found, the exception handler object invokes its handler function. COOL provides default exception handlers for the predefined exception types, such as reporting a description of the exception to the standard error stream and exiting the program or dumping a core image. A default handler is only invoked if no handler for the raised exception is found on the global exception handler stack.

The COOL exception handling macros, **RAISE** and **HANDLER_CASE**, provide the same type of functionality as the **throw** and **try/catch** statements proposed by Koenig and Stroustrup [9]. Both **throw** and **RAISE** transfer control to the most recently established handler for a particular type of exception. However, any object may be used as an argument in a **throw** expression, whereas **RAISE** only allows exception objects. In a similar manner, the **try/catch** block and the **HANDLER_CASE** macro establish handlers while executing a body of statements. The difference here is that the **catch** expression in a **try** block is like a function definition and any data type can be specified in the declaration. The case statements in the **HANDLER_CASE** macro, on the other hand, accept only COOL symbols which identify an exception type.

The differences mentioned above are minor, however, when compared to the philosophical models each system follows: termination versus resumption. In the one, the **throw** unwinds the stack before the call of the exception handler in the **try/catch**, thus supporting a termination model for exception handling, while in the second, **RAISE** expands into a function call which searches for an exception handler to invoke, thus supporting the resumptive model of exception handling.

It is interesting to note that although COOL allows both termination and resumptive models for handling exceptions, only default handlers and termination (or more appropriately, retry) handlers were used for exceptions raised in the COOL classes. Support for a resumptive model did not require much additional implementation work, but we discovered that the termination/retry model is the most appropriate for a generalized class library. A tight binding (or contract) between the class member function invoking an exception and the application function in which the exception might be resumed is absolutely necessary to ensure that *all* semantic and state information is transmitted and understood effectively by a handler. It is unlikely that this scenario is true in anything other than tightly coupled modules of a single application, which makes the usefulness of supporting a resumptive system questionable.

2.3. Symbolic Computing

COOL supports efficient and flexible symbolic computing by providing symbolic constants and runtime symbol objects [7]. You can create symbolic constants at compile-time and dynamically create and modify symbol objects at runtime by using a simple macro interface or by directly manipulating the objects. Symbols and packages are used within COOL to manage error message text for translation, to provide polymorphic extensions for object type and contents queries, and to support sophisticated symbolic operations not normally available in conventional compiled languages.

The fundamental COOL symbolic computing capability is supported through the **Symbol** and **Package** classes. The **Symbol** class implements the notion of a symbol that has a name with an optional value and property list. The name is a character string used to identify the symbol. The value field refers to some C++ object. Property lists are lists of alternating names and values which allow the programmer to associate supplemental attributes with a symbol. This property list feature has been used, for example, to easily add an international representation for all message strings to an application by representing the messages as symbol objects with the translations for different languages stored on the property list.

Symbols are interned into a package, which is merely a mechanism for establishing isolated namespaces. The **Package** class implements a package as a hash table of symbols and includes member functions for adding, retrieving, updating, and removing symbols. This package information is maintained across file module boundaries in an application-specific file, providing a crude application database for the registration of shared information. This file is used in COOL to store such things as class hierarchy information, class names, and the location of where a parameterized template class is generated. This last item is necessary in order to automate the expansion of a template exactly once within a single application. Clearly, such implementation techniques are an indication that C++ is stretching the limits of the separate compilation model of software development traditional in the UNIX® environment. We have found this type of inter-file support, from both the language and the supporting programming tools, to be absolutely necessary for the productive development of complex C++ applications using libraries of reusable components.

2.4. Runtime Type Support

COOL supports an efficient runtime type checking and query capability, and a describe mechanism for classes which derive from the **Generic** class [6]. The COOL preprocessor automatically generates for each **Generic**-derived class a list of symbols which provides the class type and class inheritance information and which is used by the **type_of()** and **is_type_of()** member functions of the **Generic** class.

The **Generic** class is inherited by most of the COOL classes. A significant benefit of this common base class is the ability to declare heterogeneous container classes parameterized over the **Generic*** type. These classes, combined with the current position and parameterized **Iterator** class, allow the programmer to manipulate collections of objects of different types in a simple, efficient, and extensible manner.

The symbols generated by the COOL preprocessor are added to a single file that functions as the application symbol repository. This file is compiled and linked with the application to allocate storage, and to initialize the symbols and the global symbol package at program startup time. An automated method for insuring correct package setup and symbol initialization is accomplished by establishing the correct dependency in an application makefile, and through global static object initialization supported by the C++ language.

The power of the symbolic computing features available in COOL significantly enhances the ability of the application programmer to solve problems in a variety of domains. Unfortunately, the complexity of the symbol system and the necessity for an application-specific database to support it has severely limited its use. We believe that many of the questions and difficulties reported to us are directly or indirectly related to this feature. The basic problem is the lack of a containing environment with knowledge about the whole application structure. This problem is reflected in the template expansion process, the runtime type system, and the symbol/package mechanism. In addition, the difficulty C++ compilers are having with enforcement of the "one-definition" rule and 100% type-safe linkage can also be directly traced to this problem. We do not believe that file-based storage repositories are the answer, no matter how much automation and "magic" is used. Fundamentally, these types of issues require a supportive environment for a robust and elegant solution, much as is found in other languages such as Lisp and Smalltalk.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

3. Class Hierarchy Overview

The COOL class hierarchy is a rather flat inheritance tree, as opposed to the deeply nested Smalltalk model. All complex classes are derived from the **Generic** class to facilitate runtime type checking and object query. Simple classes are not derived from **Generic** due to space and efficiency concerns. Each parameterized container class inherits from a base class which includes all type-independent code. The COOL class hierarchy is as follows:

- Bignum
- Complex
- Pair<*Type1,Type2*>
- Range
  - Range<*Type*>
- Rational

- Generic
  - Binary_Node
    - Binary_Node<*Type*>
  - Binary_Tree
    - Binary_Tree<*Type*>
      - AVL_Tree<*Type*>
  - Bit_Set
  - Date_Time
  - Exception
    - Error
      - System_Error
      - Verify_Error
    - Fatal
    - System_Signal
    - Warning
  - Excp_Handler
    - Jump_Handler
  - Generic<Type>
  - Gen_String
  - Vector
    - Vector<*Type*>
      - Association<*Type1,Type2*>
  - List_Node
    - List_Node<*Type*>
  - List
    - List<*Type*>
  - Hash_Table
    - Hash_Table<*Type1,Type2*>
      - Package
    - Set<*Type*>
  - Iterator<*Type*>
  - Matrix
    - Matrix<*Type*>
  - D_Node<*Type,nchild*>
  - N_Node<*Type,nchild*>
  - N_Tree<*Node,Type,nchild*>

Generic (cont'd)
- Queue
  - Queue<*Type*>
- Stack
  - Stack<*Type*>
- Symbol
- String
- Timer
- Random
- Regexp

4. Who Tried To Use COOL And Were They Successful?

A C++ class library should be targeted for a particular audience and domain in order for some measure of success to be easily determined. Initially, the COOL project had no specific customer in mind. Our project goals were to gain experience with the C++ language and determine if a rich collection of classes could be designed and used by a variety of application programmers in a practical and worthwhile manner. At the time, there were several groups in different parts of the corporation that had expressed some interest in the language. Many of these users were former Lisp programmers. Others were considering using ADA, while still others were C programmers on UNIX and PC platforms. To collect and disseminate information to this diverse user-group, we established an e-mail forum for discussion of ideas, issues, design reviews, and so forth. This provided valuable information and insight into the needs of a variety of customers. It also, however, resulted in many conflicting requests that required space/time tradeoffs in the class designs and implementations.

The potential users turned out to be programmers who were likely to use C++ and our class library in the short term and programmers who had no immediate need or opportunity for use, but were interested for possible future projects, intellectual, or personal reasons. Several projects had just begun their design and prototyping phases when they evaluated COOL. One project which was building a VHDL simulator, decided not to use COOL and developed their own C++ classes for performance and efficiency reasons. A second project which was working on an embeddable forward-chaining rules system, used our class library and the extended features such as parameterized types and the symbolic computing capability. Another project used some components of our class library, yet also designed their own versions of some classes too. In each case, there was a desire and need for many of the basic data structures found in COOL. The answer to the question "Were they successful?" is "partially". The primary reason for our lack of success was that programmer expectations, design, requirements, and the C++ language itself were not always in line with each other. The following sections contain details about which COOL components were used and how they were or were not appropriate for the work at hand.

4.1. What Did Our Users Like?

To date, the two most favorably received aspects of COOL are the implementation of parameterized templates and the portable nature of the library. The rules compiler project mentioned above extensively used not only our collection of parameterized container classes, but also wrote several application-specific template classes using the same mechanism. In general, we have received favorable response from this project concerning the syntax and expressive power of the template mechanism. Some project leaders expressed considerable willingness to give up a small percentage of performance and/or efficiency if that resulted in a highly portable software platform upon which they could design and build their applications. This position, however, was not universally shared with the engineers responsible for implementing the application.

Another strongly echoed statement is that the distribution of the class library in source format significantly enhanced the understanding of the C++ language, and the use of the classes, polymorphism, and class derivation within an application framework. Many users were in the process of learning C++ and found the ability to examine working source code for various class library components a great aid. We also found that all possible uses of a given class could not always be anticipated in our iterative design process, so that decisions such as the **private/protected** interface were often incorrectly specified. In addition, when an application uses multiple inheritance with library classes, it sometimes becomes necessary to change the inheritance specification for one or more shared base classes to **virtual**. Finally, parameterized templates can be thought of as meta-classes in that only one source base needs to be maintained to support numerous variations of a kind of class. This requires distribution of template source code unless each vendor adopts its own encrypted or partially compiled format. This, however, seems too restrictive and not desirable from the user's point of view.

We often found that users who needed a particular type of class would examine the COOL design and implementation for a similar class, then proceed to copy the source code and significantly alter and modify the interface, resulting in a class very different to that initially supplied. The reason most often sighted for this course of action was not due to functionality deficiencies or difficulty with the **private/protected** interface, but rather a perception that it must be inefficient because it was not hand-crafted by the individual. This reaction is at the heart of the problem of code reuse and whether or not programmers will accept such a course of action. It seems to depend partially upon the individual's "pain-threshold" for modifying and/or creating a new class verses the perceived value of the library class. For small-value classes such as string, the answers seems to be variable, but for larger value classes such as regular expression and text buffer, the decision is much more likely to favor using the library class [1].

We feel that for the most part, COOL provides very efficient implementations of a variety of data structures. However, the full-featured nature of the classes may be inappropriate for all users. We originally implemented one heavy-weight **String** class, for example, that contained pattern matching capabilities and implemented reference counting and other memory management techniques. We later added a streamlined version of this class that had a subset of the member functions and provided only the most basic string operations. In many situations, this was the more popular class of the two. In those cases where a more full-featured class was needed, the ability to upgrade and have a compatible interface was appreciated. A similar request was made for the **List<*Type*>** class. We therefore feel that one possible design choice is to provide two libraries (which, if we were in advertising, would be promoted as COOL and COOL-Lite). This approach seems appropriate for general purpose class libraries and could be applied to other more specific categories. For example, a database library might have light-weight classes that provide basic storage and retrieval facilities perhaps built on a flat file ISAM for speed and portability, but also offered a richer and more powerful class with concurrency control, nested transaction support, logging and recovery.

4.2. What Did Our Users Dislike?

The most common problem voiced from users concerned the requirement that any of their applications that used a COOL class derived from **Generic** required that the entire symbol and package mechanism also be linked into their application, even if they did not use the symbolic computing facilities. This is a problem inherent with class hierarchies and libraries with complex or intertwined dependencies. For example, a class derived from **Generic** will result in implementations of the templates for **Hash_Table** and **Package** also being linked into the executable image. An additional concern already mentioned is the complexity of the symbol setup and the necessity for an auxiliary database file to be compiled and linked with each application. This is partially due to our implementation and the necessity for portability, but also because of the state of current linker technology on many platforms. Many commercial vendors implementing environments should not have this problem.

Another significant difficulty was the communication between the library developers and the library users on the intended use of and interface for the classes. A C++ class browser utility would greatly simplify the problems of educating a programmer about the available classes and their functionality. As class libraries grow and the relationships between objects become more complex, the usefulness and applicability of traditional tools such as **grep**(1) and **more**(1) begin to break down. More modern tools designed for this problem such as the graphical class browser in the Saber C++™ development environment will substantially ease the learning curve and information explosion. Using such tools, a programmer will be able to more easily identify inheritance problems, locate state and member function definitions, and assimilate a more complete mental model of the library. This will be particularly true if in fact the promise of integrating several class libraries for different components within a single application is to be realized.

5. What Did We Learn and What Would We Do Differently?

With what we now know, we would make several different design decisions, the first and foremost of which would be to simplify the interdependencies between the classes. This would be accomplished by essentially flipping the class hierarchy. Instead of having a base class **Generic** that provides the run time typing capability and from which most other classes are derived, we would provide this class as a standalone class. Classes such as **String** and **Vector<*Type*>** would not be derived from **Generic**. In this manner, users who wanted the functionality of one or more objects found in COOL would not get runtime type capability linked into their application. Those users who needed the symbolic computing facilities could use multiple inheritance to derive the appropriate class. For example, a **String** class with type-query support could be multiply derived from **Generic** and **String** to produce the desired result.

The **Exception** and **Excp_Handler** classes are the only classes in COOL which require the runtime type checking capability and symbol and package mechanism. This results from our current implementation of handling exceptions. We would change this implementation and use simple character strings instead of COOL symbol objects to represent each exception class name. Most of the COOL classes raise exceptions and we would still want to eliminate the necessity of including the symbol and package mechanism when using any of these classes.

In addition to removing the **Generic** dependency in all classes, we would also provide a

Saber C++ is a trademark of Saber Software, Inc.

common base class for the container classes that support the notion of an iterator object to allow for the commonality of these class objects. This base class would probably contain only pure virtual member function specifications to enforce a particular interface in the derived classes.

Finally, we would also provide both a simple version and full-featured version of many of the classes. For example, we would implement a version of the parameterized classes which restricts the type parameter to data type pointers and removes the use of references. We believe that providing the simple version of classes would satisfy many users who would otherwise alter a COOL class implementation to reduce its complexity for space/time considerations. In addition, it would simplify the design for the COOL classes making them easier to use, and allowing for more code reuse.

We like the "forest" hierarchy structure and believe that it is more appropriate for applications written in C++ than the Smalltalk deeply nested structure found in other class libraries, such as Gorlen's NIH class library [8]. In an effort to reduce even further the complexity and size of an application that uses the class library, we would also opt for placing every non-inline member function in its own source file and the resulting object file in the archive. This would force linkers to link only those member functions actually utilized in an application into the executable image. Finally, we are concerned about the single namespace and the inevitable name clashes that result when two independently developed class libraries are combined in a single application. In one case we are familiar with, a user trying to use COOL with the Stanford InterViews class library [10] had to make several changes to class names and functions that were common to both libraries. It appears that with the current language definition, the only reasonable solution is to require prefixing all global names with a two or three letter prefix in order to reduce the chance of a clash. This, however, is clearly not acceptable and we believe a language extension to isolate namespaces is the only viable long-term solution.

6. Conclusion

The COOL project has been an exciting and rewarding experience, serving not only to fulfill our initial intent of providing valuable experience with C++, but also as a focal point for a larger discussion within the company regarding software productivity and code reuse. Our diverse user-group has provided valuable information concerning applicability and potential for utilization of C++ class libraries in a variety of projects and platforms. We conclude that a class library consisting of many basic data structures and templates:

- can significantly aid the portability of an application
- will be viewed with considerable skepticism by many C programmers
- must be supplied in source code format
- will often be modified/decomposed to suit the purpose
- provides a medium for the dissemination and spread of ideas and techniques

While most applications require many of the basic data structures found in COOL, there are always other necessary components. It is our conjecture (based on a limited application test set), that many projects will actually require four types of class components:

- basic data structures -- strings, templates, containers, etc.
- user-interface widgets -- menus, buttons, dialog boxes, etc.
- network/communication -- file transfer, TCP/IP, remote access, etc.
- application-specific -- domain-specific "high value" objects

If there is one significant lesson we have learned from COOL, it is that W.C. Fields was right in saying: "You can't satisfy all the people all the time!" On the other hand, a C++ data structure class library organized in a "forest" hierarchy has components that can, in combination with other libraries, satisfy most of the people most of the time. We think this is about the best that is possible.

7. Status

COOL is currently running on a Sun SPARCstation™ 1 running SunOS™ 4.x, and a PS/2™ model 70 running OS/2™ 1.2. The SPARC port utilizes the AT&T C++ translator (cfront) and the OS/2 port utilizes the Glockenspiel C++ translator (which is a port of the AT&T translator) with the Microsoft C compiler.

8. Acknowledgements

Many people contributed ideas, suggestions, and criticisms that have helped shape the class library evolution and development. Amongst these are Fred Burke, Terry Caudill, Merrill Cornish, Carey Jung, Asif Malik, Dane Meyer, LaMott Oren, Jeri Steele, Dan Stenger, and Brian Victor.

9. References

[1] James Coggins, *Design Criteria for C++ Libraries,* USENIX C++ Conference, San Francisco, CA, April 1990.

[2] Andy Daniels and Kent Pitman, *Common Lisp Condition System Revision #18,* ANSI X3J13 subcommittee on Error Handling, March 1988.

[3] Margaret Ellis and Bjarne Stroustrup, *The Annotated C++ Reference Manual,* Addison Wesley, 1990.

[4] Mary Fontana, Martin Neath and Lamott Oren, *A Portable Implementation of Parameterized Templates Using A Sophisticated C++ Macro Facility,* Information Technology Group, Austin, TX, Internal Original Issue April 1990.

[5] Mary Fontana, Martin Neath and Lamott Oren, *A Portable Exception Handling Mechanism for C++,* Information Technology Group, Austin, TX, Internal Original Issue April 1990.

[6] Mary Fontana, Martin Neath and Lamott Oren, *A Runtime Type Checking and Query Mechanism for C++,* Information Technology Group, Austin, TX, Internal Original Issue November 1990.

[7] Mary Fontana, Dane Meyer, Martin Neath and Lamott Oren, *Symbols and Packages in C++,* Information Technology Group, Austin, TX, Internal Original Issue November 1990.

[8] Keith Gorlen, *An Object-Oriented Class Library for C++,* USENIX C++ Workshop, Santa Fe, NM, November 1987.

[9] Andrew Koenig and Bjarne Stroustrup, *Exception Handling for C++,* Submitted as document X3J16/90-042 to the ANSI C++ committee, July, 1990.

[10] Mark A. Linton, Paul R. Calder, and John M. Vlissides, *InterViews: A C++ Graphical Interface Toolkit,* Technical Report CSL-TR-88-358, Stanford University, July 1988.

SunOS and SPARCstation 1 are trademarks of Sun Microsystems, Inc.
PS/2 is a trademark of International Business Machines Corporation.
OS/2 is a trademark of International Business Machines Corporation.

[11] Mike Miller, *Exception Handling Without Language Extensions,* Proceedings of the USENIX C++ Conference, Denver, CO, October 17-21, 1988, pp. 327-341.

[12] Bjarne Stroustrup, *Parameterized Types for C++,* Proceedings of the USENIX C++ Conference, Denver, CO, October 17-21, 1988, pp. 1-18.

[13] Texas Instruments Incorporated, *COOL User's Guide,* Information Technology Group, Austin, TX, Internal Original Issue January 1990.

Pragmatic Issues in the Implementation of Flexible Libraries for C++

Bruce Cohen (brucec@crl.labs.tek.com)
Douglas Hahn (hahn@crl.labs.tek.com)
Neil Soiffer (soiffer@crl.labs.tek.com)
Tektronix Labs
Beaverton, OR 97077

Abstract

Libraries designers are traditionally faced with making a number of tradeoffs involving speed, space, functionality, and ease of debugging. Because no two users have the same needs, the library design is likely to displease every user in some way. C++, through the use of inheritance, virtual functions, inlining, etc., offers many opportunities to appease a greater number of users. Unfortunately, the use of these features can significantly effect the performance of the code. In this paper, we present a number of techniques and tools that can be used to decrease linked code size by as much as 90% or to increase speed by as much as a factor of three. These techniques and tools have application to any reusable code.

1 Introduction

Users are not always interested in the fastest possible library code, especially if that speed comes at the expense of needed space or debugging ease. A case in point is that of libraries designed for embedded systems. Embedded systems range from hand-held systems where space and power place severe limitations on the amount of memory that can be used (for both code and data) to high-end systems where speed is the primary factor and memory costs are not very important.

In order to satisfy the greatest number of users, a library implementation should allow users to configure the library according to their needs. In particular, users should be able to choose from a broad range of functionality in the interface and also optimize for:

- **space** either code or data
- **speed** as close to tuned C code speed as possible
- **debugging** having argument checking, minimizing the impact of inlining

Unfortunately, these are often conflicting goals. Our library implementation uses a variety of techniques to allow users to pick and choose what sorts of optimizations are made. The techniques discussed in this paper are applicable not only to library code, but to any reusable code written in C++. An additional concern raised by the use of C++ is the time and space efficiencies involved in initializing static data. We further discuss this problem and our solution in Section 3.7. We believe that application of these techniques can result in object code that is competitive in either speed or space with application-specific code.

This paper begins with a overview of our library implementation. Section 3 deals with various speed or space bottlenecks and the techniques we use to overcome them. Finally, we discuss some of the challenges of testing flexible libraries. The speed and size measurements given in this paper are discussed in the appendix.

2 Basic Library Design

Our library is designed to be a reusable component in both general workstation C++ work and in embedded designs. Currently, it consists of *abstract data types*—abstract functional entities such as sets, queues or graphs as opposed to what we call *data structures*, such as linked lists or hash tables that have well understood space/time properties. Our design goal is to provide a broad high-level library interface that is typesafe, encourages reusability and allows portability. We consider speed and space considerations an essential part of reusability—if a library component is not on par with application specific code, projects will want to reinvent the wheel.

Our library consists of a parameterized forest similar to the GNU C++ library[Lea88]. Most trees in the forest consist of a root data type that has different implementations derived from it, each containing a data structure. For example, derivations of the data type `Set` are implemented via fixed-size arrays, linked-lists, and hash tables (among others). Polymorphism is thus supported only among different implementations of a particular data type. In addition to allowing differing implementations, abstract data types enhance "plug-compatibility" and standardization of interfaces.

Parameterization is done via templates; a tool called *gen* performs sed-like textual substitution to instantiate all necessary source files for a particular data type as well as a make file to compile them. The proposed ANSI standard template mechanism[ES90] was deemed inadequate because it does not handle three important problems: shared implementations, flattening, and value versus reference parameters. These are discussed in Section 3.

Providing a number of implementations for each data type allows users to choose the general space/time tradeoff that is appropriate for their application. Inheritance substantially reduces the amount of code that the library implementor must write. It can also reduce code space if a user needs two different implementations of the same data type (e.g, an initial phase of the program may perform many additions, later phases only need to access elements; different implementations for the different phases may be appropriate). Additional space savings can be achieved by users of several data types by choosing the same underlying data structure for each type, thereby reusing the data structure code.

Two important issues not yet addressed by our library are memory management and concurrency. Flexible memory management policies and concurrency strategies as provided for in the Booch components[BV90] allow libraries to be used in an even greater number of applications.

3 Techniques for Obtaining Flexibility

In this section, we describe space or time bottlenecks we encountered in our C++ library implementation, and how the design, C++, or the UNIX environment exacerbated them. We also detail some solutions to these bottlenecks. A number of these solutions involve simple use of `#defines`. For example, `#defines` are used to control whether argument checking is performed or not. While this may seem to be an obvious idea, none of the three C++ libraries that we looked at (NIH library[GOP90], et++[WGM88], GNU libg++[Lea88]) used these ideas. Instead, argument checking is either always performed or never performed and can not be altered by a user.

Embedded systems programs may require fine-grain control of the placement of data in memory. We have developed tools to help do this. These tools also help in optimizing the initialization of objects in embedded and non-embedded systems.

3.1 Argument checking

Extensive argument checking is essential to faster debugging and program development. However, by the time a product is tested and ready to ship, bugs detectable by data structure argument checking have hopefully been eliminated. Turning off the argument checking via compile-time switches allows both an increase in speed and a reduction in code size. In addition, some data that speeds error checking (e.g., handshakes) and functions can be eliminated when argument checking is turned off, thereby decreasing data space.

Measurements on our library indicate speedups of two times[1], although speedups vary greatly with the nature of the checks. Code size reductions of 10% (without flattening) and 25% (with flattening) were achieved.

3.2 Inlining

A common perception is that inlining substantially increases speed at the expense of code space. However, we have found in our implementation that there is a *decrease* in the code space if only the simplest of member functions are inlined—functions whose code size is equal to or less than the amount of code needed to perform the function call. Since the function is inlined, the code for the stand-alone function is never emitted, resulting in a net decrease in code space. Hence there is a notion of inlining for space, which reduces code size while providing a substantial increase in speed, and inlining for time, which results in a further increase in speed at the expense of an increase in code size.

The user may not wish to inline at all if it interferes with debugging the system. To accommodate the user without increasing code size, it is necessary to organize the source files into more than two files. We divide all code into three files: a header file (.h), a source file (.C), and an inline file (.i). At the end of the .h file, the .i file is included using

```
#define DOTCFILE 0
#include class.i
#undef DOTCFILE
```

while at the beginning of the .C file, the .i file is included using

```
#define DOTCFILE 1
#include class.i
#undef DOTCFILE
```

Inside the .i file, member functions are indicated as being inlined for space or time:

```
#if (SPACE ^ DOTCFILE)
SPACEINLINED void classname::member ()
{
    return 1;
}
#endif (SPACE ^ DOTCFILE)
```

Space inlining yields speedups of 5.7 or 1.7 times and size reductions of 30% or 15% for flattened or non-flattened Sets respectively. Flattening yields greater speedups because a large number of simple access functions are virtual; flattening removes their virtualness. This removes a function call and allows the compiler to make contextual optimizations. Time inlining yields similar speedups (on even more of the operations) and reductions in size of the Set code, but substantially increases the size of the user code.

3.3 Shared Implementations

To instantiate a class, the parameters are replaced by actual type names—creating type-safe source code. For a data structure instantiated with a pointer type, this leads to identical code, simply different pointer type names. To avoid this replication, a technique suggested by Stroustrup[ES90] and others of writing type-safe *cover classes* is used. Cover classes are classes that are subclassed from a void* instantiation of the parameterized class; they perform explicit typecasts to provide a

[1]All times are for the `apply` function, which is implemented in the `Set` base class. The space measurements are based on the change in the code size of the `Set` classes. See the appendix for details.

type-safe interface for pointer types. All functions defined by the cover class are inlined for space and cause no time or space degradation. *Gen* automatically generates the proper cover classes and base classes when instantiating pointer types. A preferable approach is to have the C++ compiler perform the type-checking for different pointers, but only emit one copy of generalized code.

3.4 Value vs. Reference Parameters

When implementing a function, the implementor must decide whether the parameters should be passed by value or by reference. This decision mostly affects time, but copying even moderately-sized structures can have a space impact also. The library designer cannot know whether the objects passed in are large or small, and hence, what is the appropriate mechanism to use. The instantiation program *gen* that we developed solves this problem by allowing the user to specify whether to use value or reference semantics. The library code uses <T&> parameters that are filled with "type" or "type&" depending on whether value or reference semantics are chosen.

With the proposed ANSI template mechanism, this can be done in a clumsy manner by doubling the number of parameters to a template[Lea91]. For example, the following template allows the user to decide whether objects are passed by reference or by value.

```
template <class T, class Targ> class TemplateExample
{
    void add(const Targ element);
    T    remove();
}
```

Reference semantics are achieved by using a reference type for the second argument as illustrated below.

```
TemplateExample(int, int)                // pass by value
TemplateExample(Window, Window&)         // pass by reference
```

Measurements using `ints` indicate a 25% speedup by using pass by value instead of pass by reference semantics for simple access functions.

3.5 Flattening

Virtual functions basically have three uses. One use is for polymorphism. A second use of virtual functions is to allow extensions or modification of functionality of a class via derivation. A third use is that of code sharing; by writing functions that take a base class as an argument, the code need not be replicated for each of the derived classes (unlike in a language that provides only genericity[Mey86, Ada83]). Unfortunately, using virtuals has the serious side effect in C++ of preventing inlining of many time-critical functions,[2] since the compiler cannot determine at compile-time that a class is not derived from elsewhere in the system. This could be circumvented through the use of customization facilities[Lea90] or via advanced link/loading techniques[Joh90] (and cf. Ada inlines at link-time[Ada83]).

The approach we took was to build a primitive flattening tool, that a user who does not use the code-sharing or polymorphic aspects of a data type can use to generate a *flattened* class instead of a derived class. A flattened version of a class hierarchy $A_1, \ldots, A_n$ is a single class A' containing all of the functionality of $A_1, \ldots, A_n$, but containing no virtual function calls. Ideally, this flattening should be done fully automatically. Currently, our simple prototype merges header files, source files, and inline files, eliminating virtual functions definitions and renaming all references to the base and derived class to a neutral new classname (e.g., FlatHashSet). Redundant function definitions (i.e., those functions defined in both the base and derived classes) and constructors are handled manually by using `#defines` in the source code. Typically, there is at most one `#define`/file needed.

[2]Typically, the simplest functions, such as access functions, are the functions that must be virtual.

Because the current primitive state of the flattener only allows us to easily flatten two classes, we compressed the library hierarchy to be only two levels: a higher abstract class such as `Container` exists in name only and its functionality is implemented for each data type as if they were derived from it. An actual base container would cause almost all functions to be virtual. We felt that the increase in speed is worth the lost potential polymorphism and increase in space.

Measurements indicate speedups of three times and size reductions of 40% by using flattening.

3.6 Splitting

A problem incurred by the use of C++ under UNIX in designing a reusable library is the fact that classes are organized as a single .o file: if any function in that .o file is used, all of the functions in the .o file are included in the linked image. Some people such as Coggins[CB89] have suggested complicated schemes of arranging libraries that involve writing only one member function per file. These schemes can be very cumbersome to the library implementor. Even if each member function is placed in its own .o file, all virtual functions will be referenced from the virtual function table (which is replicated in each of the .o files), when many of those functions are unused by the program. A much more effective method would be to have the C++ compiler output a symbol descriptor telling which vtable and slot is being used for each virtual invocation. Then at link time, all unreferenced vtable slots are 0'ed (taking multiple inheritance into account) and all unreferenced member functions are eliminated.

For prototyping, we take the assembly language output of the compiler and split it into multiple components, renaming static global variables as necessary. We then assemble those components and create an archive. This allows the UNIX linker to pull in only those functions that are needed (though users must often specify the archives more than once because of linker ordering rules). Splitting is done at the assembly language level because it simplifies some scoping issues. Splitting causes debugging information to be duplicated, resulting in an explosion in the size of the symbol table. Because of this and because splitting takes much longer than normal compiling, splitting is typically done after the debugging phase.

A problem not solved by this approach to splitting is that unused virtual functions are still referenced. However, with flattening available, this does not appear to be an excessive problem.

The amount of space savings due to splitting depends upon the many factors including the breadth of the library interface (i.e., does the library include the kitchen sink) and how much a program exercises the interface. For our test program (which exercised a modest cross section of the interface), splitting reduced `Set` code size by 70%.

3.7 Object Initialization

Program initialization in C++ is much more complex than with C because of the (lack of) order of construction of static objects and difficulties generating compile-time static initializations of objects. This is especially true for objects of classes with virtual functions.

Initialization is a particularly acute issue with embedded systems. This is because the economics of using Read-Only Memory (ROM) versus Random-Access Memory (RAM) often make it undesirable to dynamically allocate objects. Static allocation and initialization can remove memory-management space overhead and the need for initialization code for the data. Reducing the amount of initialization code also reduces startup time. Unfortunately using a static constructor in C++ does not guarantee that the object will be constructed as constant data[3]: the compiler is free to generate code to initialize the object instance. This means that an object may not necessarily be loadable into ROM just because a static constructor is used. Also, many otherwise constant objects are often algorithmically initialized because of complex interdependencies between objects that are hard to express at compile-time. Often these dependencies, as useful as they are to the programmer, would not have been used if C++ did not make them so easy to manage.

The ability to run a program, stop it, and restart it later already exists in UNIX. However, this image contains unnecessary initialization code and a fragmented heap. For embedded systems

[3]Far from it; the current compilers won't allow it at all in most cases.

where space is limited, the wasted space of now-unneeded initialization code and heap overhead is undesirable. Furthermore, because it is not possible to distinguish the newly constant data from the nonconstant data, all of the data must be copied from ROM to RAM at startup. We are working on a tool that allows users to "clone" a program from the initialized image. Cloning can be thought of as a mechanism for advancing the time of binding of an object's value from run-time to build-time.

Currently, we are experimenting with two techniques for image cloning that differ in how they get object structure layout information and how they merge the initialized objects in with the application program. To clone an initialized object, all of the values of its members must be copied. Any members whose values are pointers must be identified so that the pointer values can be modified to account for differences in placement of the objects between the initialization program and the final build of the application program. Virtual table pointers are a special case of this pointer member problem. The information that must be known about an object to be cloned includes:

- the type (class) of the object,
- the base address of the object,
- the offsets, types, sizes (for arrays), and values of all members in the object and the placement of all virtual table pointers.

In what we call "internal source cloning," instrumentation code similar in concept to that used by Dossier[IL90] is added to the initialization code of the application program; this initialization code is typically kept in a separate file for the sake of clarity. The user's class inherits this instrumentation code along with a boiler-plate class field description to provide run-time type information about itself. Currently, the boiler-plate is hand written, but we plan to move to an automated source transformer to add the information. The instrumentation classes add a metaobject protocol modeled on a subset of the CLOS MetaObject Protocol[CLO88]. Run-time class-description objects keep track of the internal structure of each class of objects that can be cloned, and each object to be cloned keeps a reference to its class object. The metaclass instrumentation can be added on an application-class-by-class basis, and individual objects can be tagged as being clonable when they are allocated in the application program. When the program is compiled with this instrumentation and run, the instrumentation records the creation of objects and writes initialized structure declarations of the objects out to a file as C structures (not writing C++ source out allows us to initialize the virtual table pointers). Each object is given a unique name and all pointers to cloned objects within other cloned objects are recorded symbolically using the name of the structure pointed to. This allows the linker to resolve the addresses of the references later. The output file is then compiled as part of the build of the final program, with the objects being referenced with external global linkage rather than as dynamic objects. This technique allows building the cloned object records on any development system and compiling for any target system for which a cross-compiler is available, a useful capability in building embedded software to be ported across multiple target architectures.

The internal cloning instrumentation was implemented using a very general metaobject model so that we could extend it easily. In particular, we intend to add the ability to specify transformations on objects as they are cloned. The application programmer could write his code using growable containers, for instance, allowing them to expand to whatever size is required during initialization, and have them mutated into fixed-size containers when they are cloned, so as to save the data overhead in the container object. We also wanted to use the cloner to investigate the utility of metaobject protocol instrumentation. In the future we hope to apply the technique to other applications, such as creating wrappers for accessing remote objects.

In "external a.out cloning," the initialization portion of the application program is compiled with debugging turned on and the program is run to the completion of initialization. The program uses an overloaded new that records the textual name of the class at allocation time. Once initialization is complete, the program's symbol table is read and the heap is traversed. For each member that is a pointer (as determined by the symbol table object description, or in the case of unions, a user supplied tag resolver), the object pointed to is determined, and a relocation record created for that pointer. A compacted version of the heap along with the set of relocation records is written out as a clone.o file. The clone.o file is linked with the runtime version of the program.

Compared to the internal technique, the external technique generates the cloned objects with less work on the part of the application programmer, at the expense of producing processor-architecture-dependent object descriptions. Also the external technique does not allow higher level optimization of objects. However, it is less tied to the details of the compiler, such as name mangling.

4 Testing Flexible Libraries

A downside of the flexibility that we advocate is an increase in the complexity of testing. The library must be tested for different instantiations of the parameters, for each of the different implementations of a data type, and for each of the various configurations (e.g., inline time/space/off, flatten on/off). While there is no substitute for testing each of the configurations, much of the test code can be reused.

An object-oriented approach to testing allows us to write generic tests for containers and data types; implementation independent test code (black box tests) are written for each level of the inheritance tree. For example, all containers in our library have the functions: `size`, `removeAll`, `isEmpty`, and `includes`. The container testing code is called with a container, a size, an element in the container, and an element not in the container in order to test these functions. Similarly, all `Sets` have the functions `subset`, `add`, and `remove`. The `Set` testing code creates two `Sets` and several distinct elements. Elements are added to and removed from the `Sets` to test these functions. The `Set` testing code calls the container testing code to verify that the container functionality works. `HashSets` are an implementation of `Sets` and add constructors to `Sets`. The `HashSet` testing code creates several `HashSets` and calls the `Set` testing code. If `HashSets` redefined some `Set` or container functions, these would have to be tested by the `HashSet` testing code also. The `HashSet` testing code also uses knowledge of the implementation to test possible trouble spots such as adding enough elements to force the hash table to resize.

The test code is parameterized and is instantiated by *gen* for the following types: `int` (a built-in type); `Obj` (a user-defined type, passed by value); `Obj&` (a user-defined type, passed by reference); and `Obj*` (a user-defined type, passed by pointer). A make file is used to control these instantiations along with compiling code with alternative defines, flattened versions, etc.

A problem with parameterization that we have not seen mentioned in the literature is that instantiation of a parameter can result in ambiguity of overloaded functions. The ambiguity occurs in two circumstances: instantiation with similar types and instantiation with types similar to default types. The following example demonstrates the problems when `<T>` is instantiated with `int`.

```
class <T>Conflict
{
    <T>Set container;
  public:
    <T>Conflict(int size=4);
    <T>Conflict(<T> element, int size=4);
    foo(int);
    foo(<T>);
};
```

The constructor call `intConflict(10)` is ambiguous. Testing exposed these problems.

5 Conclusions

To summarize the techniques described above, some involve using features of the language (such as argument checking and inlining for time or space, while others use separate tools (flattening, splitting and cloning). These techniques are necessary to eliminate bottlenecks caused by:

- **design requirements** argument checking, polymorphism and complex initialization
- **language** no customization, inadequate code sharing (cover classes), static initialization limitations, and limited templating.
- **operating system** poor linking
- **embedded target system requirements** memory layout constraints

The amount of optimization possible (taken together the techniques can reduce code space by about 85% and increase speed by about ten times), strongly argues that C++ needs a standard set of optimizing tools, and a recommended implementation style available to end users.

In addition, extensions to the language, such as better control over static initialization, the ability to specify value versus reference semantics in templating and the inclusion of customization[Lea90] would make not just C++ libraries, but all C++ resuable code much more appealing to application writers faced with severe speed and space constraints.

References

[Ada83] *Reference manual for the Ada programming language*, 1983.

[BV90] Grady Booch and Michael Vilot. The Design of the C++ Booch Components. In *Proc. OOPSLA '90*, pages 1–11. ACM, 1990. Also available as SIGPLAN NOTICES, 25(10), October, 1990.

[CB89] James Coggins and Gregory Bollella. Managing C++ Libraries. *ACM SIGPLAN Notices*, 24(6):37–48, June 1989.

[CLO88] *Common Lisp Object System Specificaton*. ACM SIGPLAN Notices, September 1988. Special Issue.

[ES90] Margaret Ellis and Bjourne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[GOP90] Keith Gorlen, Sanford Orlow, and Perry Plexico. *Data Abstraction and Object-Oriented Programming in C++*. John Wiley & Sons, 1990.

[IL90] John Interrante and Mark Linton. Runtime Access to Type Information in C++. In *Proc. USENIX C++ Conference*, pages 233–240, April 1990.

[Joh90] S. C. Johnson. Postloading for Fun & Profit. In *Proc. USENIX Conference*, Winter 1990.

[Lea88] Douglas Lea. libg++, The GNU C++ Library. In *Proc. USENIX C++ Conference*, pages 243–256, October 1988.

[Lea90] Douglas Lea. Customization in C++. In *Proc. USENIX C++ Conference*, pages 301–314, April 1990.

[Lea91] Doug Lea. Personal communication. February 1991.

[Mey86] Bertrand Meyer. Genericity versus Inheritance. In *Proc. OOPSLA '86*, pages 391–405. ACM, 1986. Also available as SIGPLAN NOTICES, 21(9), September, 1986.

[WGM88] André Weinand, Erich Gamma, and Rudolf Marty. ET++—An Object-Oriented Application Framework in C++. In *Proc. OOPSLA '88*, pages 46–57. ACM, 1988. Also available as SIGPLAN NOTICES, 23(11), November, 1988.

%speed increase						
argument checking	on			off		
inlining	none	space	time	none	space	time
add	9	25	0	0	12	6
test	4	22	-5	-5	0	11
access	2	21	13	13	633	685
apply	20	21	13	3	242	209
reduce	9	10	86	18	74	116
remove	0	18	2	0	0	7

%size decrease						
argument checking	on			off		
inlining	none	space	time	none	space	time
stringsize	0	0	0	0	0	0
setsize	27	33	3	29	43	41
testsize	1	3	11	1	5	16

Figure 1: Effects of Flattening

Appendix: Measurements

The numbers in Figures 1–4 were obtained by running each operation a thousand times and subtracting the loop overhead. All files were compiled with "g++ -O" on a Sun3. Figure 1 shows the percent speed increase and percent code size reduction achieved by flattening under various configurations of argument checking and inlining. Similarly, Figure 2 shows the percent speed increase and percent code size reduction achieved by turning off argument checking; Figure 3 concerns inlining in the presence of argument checking; and Figure 4 concerns inlining without argument checking. The tests involved:

- **add** The cost of adding 1000 unique strings (from a dictionary) to an empty set.
- **test** The cost of testing the presence of the same 1000 words (after shuffling) in the set.

%speed increase						
flattening	on			off		
inlining	none	space	time	none	space	time
add	5	4	13	15	16	6
test	5	-1	17	14	21	0
access	91	1066	971	73	93	54
apply	55	385	97	82	66	77
reduce	61	122	58	50	41	36
remove	7	2	11	7	20	5

%size decrease						
flattening	on			off		
inlining	none	space	time	none	space	time
stringsize	17	21	20	17	21	20
setsize	13	25	48	9	11	14
testsize	0	1	9	0	-1	9

Figure 2: Effects of Argument Checking

%speed increase				
inlining	space		time	
flattening	on	off	on	off
add	176	142	141	164
test	155	118	119	140
access	214	164	193	164
apply	82	87	342	102
reduce	59	58	189	69
remove	154	115	130	125

%size decrease				
inlining	space		time	
flattening	on	off	on	off
stringsize	16	16	26	25
setsize	20	13	-4	21
testsize	0	-2	-39	-64

Figure 3: Effects of Inlining with Arg Checking

- **access** The cost of retrieving the 500^{th} word added to set using a index returned by a previously called find operation.
- **apply** The cost of applying a null function to every word in the set.
- **reduce** The cost of applying a test to all words and reducing over those words testing true (every other word) using the identity function.
- **remove** The cost of removing all 1000 words from the set by name (after shuffling).

The size reductions listed are:

- **stringsize** The object file size of the string class. Strings do not use virtual functions, but do implement storage sharing and other features that make them a little slower than using `char*`.
- **setsize** The object file size of the set class.
- **testsize** The object file size of the test module.

%speed increase				
inlining	space		time	
flattening	on	off	on	off
add	173	143	169	143
test	142	130	144	110
access	1816	195	1543	136
apply	471	71	419	86
reduce	119	48	183	54
remove	141	140	136	120

%size decrease				
inlining	space		time	
flattening	on	off	on	off
stringsize	20	20	27	27
setsize	31	15	38	25
testsize	1	-3	-27	-50

Figure 4: Effects of Inlining without Arg Checking

A Network Toolkit

Walter Milliken
(milliken@bbn.com)

Gregory Lauer
(glauer@bbn.com)

Bolt Beranek and Newman Inc.
Cambridge, Mass.

Abstract

The Network Toolkit is being developed to support experimenters who are implementing and testing advanced network algorithms. The network toolkit supports experimentation by providing classes that correspond to objects typically used in constructing network applications (packets; packet handlers; queues; etc.). The objects are simple to interconnect, so that experimenters can focus on new algorithms rather than "boilerplate" software. In this paper we discuss the design goals we established for this project and the progress we have made toward accomplishing them. We also discuss C++ issues we've encountered while implementing this toolkit.

1 Introduction

In this paper we discuss our efforts to build a C++ toolkit for developing packet-switches (routers, gateways, etc.)[1] under the Modular Tactical Gateway (MTG) project. In the next subsections we discuss the software goals of this project and overview the rest of the paper.

1.1 Goals

The main goals of the MTG project are to build an object-oriented *toolkit* for developing packet-switches (and possibly other network applications) and to use it to implement an advanced tactical gateway. In this paper we will only discuss the toolkit issues and we will not discuss the advanced tactical gateway algorithms. The main purpose of the toolkit is to simplify implementing and experimenting with new network algorithms. There are four sub-goals that follow from this main goal:

- It should be easy to implement new algorithms, thus our design decisions have favored ease of implementation rather than efficiency of implementation.
- The toolkit should be platform independent (portable) so as to make it available to the widest possible audience.
- The toolkit should be parallelizable. Since the toolkit it not optimized for efficiency, it should allow an experimenter to use more hardware to improve performance.
- The toolkit concepts, classes, and documentation should be clear so that the toolkit is easy-to-use.

[1]Work has been funded by DARPA under contracts MDA903-89-C-0020 and N00140-87-C-8910. Approved for public release. Distribution unlimited.

1.2 Overview of Paper

The rest of the paper will have the following structure:

- Section 2: Design Techniques. In the next section we discuss the design techniques we used in developing the MTG toolkit. This discussion will touch on the criteria we used to evaluate different architectures, the types of objects composing the toolkit, how we support parallelism, and how we anticipate connecting the objects together to form an application.
- Section 3: Data Classes. This section discusses the classes that represent buffers, packets, and packet handlers. It also covers our facilities for packet-level network debugging and special-purpose monitoring.
- Section 4: Control Classes. This section discusses briefly our thoughts on classes for representing control algorithms such as routing, congestion control, etc.
- Section 5: Gateway Design Example. In this section we introduce an example of a simple gateway to illustrate how the toolkit can be used to put together a complete application.
- Section 6: Language Issues. This section discusses some issues we've encountered in developing the toolkit using C++.
- Section 7: Conclusions. This section discusses the current state of the project and our plans for the future.

2 Design Techniques

In the following sections we discuss the techniques that we are using in developing the MTG toolkit.

2.1 Read-only Extensibility

Since the MTG toolkit is intended for wide distribution, it is desirable that the toolkit code itself be *read-only* from the implementor's point of view. That is, the implementor of a gateway using the MTG toolkit should not have to modify any of the source code in the toolkit to get the desired functionality. Instead, functionality should be added by subclassing the toolkit classes.

Unfortunately, subclassing is an inappropriate technique when the toolkit user desires to modify the functionality of a base class from which many other toolkit objects have been derived. The `Packet` class provides a classic example of this issue. To add new functionality to the `Packet` class requires either that the application programmer re-derive every toolkit-provided class derived from `Packet`, or that he modify the base class. This leaves him vulnerable to code maintenance problems when new MTG toolkit releases modify the `Packet` class internals — any local changes made would have to be merged into each new release.

We considered several possible designs which would alleviate this problem. One design involved having the root class contain a pointer to an auxiliary object which the toolkit user could customize — the MTG distribution would always contain an empty definition, allowing the programmer to safely define any new functionality needed. This approach adequately handled adding data elements and non-virtual functions to the base class, albeit with some clumsiness (since the auxiliary structure has to be accessed explicitly). However, this approach becomes much too cumbersome when adding virtual functions to the base class.

An alternative approach we considered involved requiring every class derived from a base class such as `Packet` to multiply-inherit from an auxiliary object class. We rejected this approach since it added additional constraints on the subclass designers, which seems undesirable.

The approach we eventually selected involves introducing an empty, user-customizable, root class, from which the toolkit-supplied base class inherits. In the toolkit distribution, this class does nothing, but a programmer can create a custom version of the toolkit base class with additional functions and data, transparently, and without risk of update problems. The result then is a class hierarchy which is safely customizable on both ends — required functionality can be added to all classes by modifying the customization root class, or new leaf subclasses can be created to add type-specific modifications.

Since we are still in the early stages of working with the toolkit, and, as the toolkit implementors, don't need to use the customization facility, we do not yet have any experience on how useful this customization mechanism will be. One clear drawback to this approach (not shared by the auxiliary object pointer approach we first looked at) is that the toolkit user must have source code access and must be able to recompile the toolkit library if he wants to use the customization facility. Since we plan to distribute the MTG toolkit in source form, this isn't a problem for us, but may limit the general utility of the technique.

2.2 Library Issues

In [3] persuasive arguments are given why libraries shouldn't be *comprehensive, monolithic hierarchies* or *a toolkit of tiny classes*. We have thus chosen to implement the MTG toolkit as a "small forest of large trees" (i.e., as a relatively small number of classes that can be readily understood by the user and which correspond to large useful concepts). As part of this approach, we are striving to keep the hierarchy shallow so as to improve its understandability. Figure 1 shows many of the major objects implemented so far in the project.

A shallow hierarchy with a few major root classes also facilitates the toolkit user's customization, as described above. Customizable root classes would be harder to implement in a monolithic hierarchy.

2.3 Protocol Layers and Functional Decomposition

To make the toolkit useful, it must support the way people think about protocols. The most common abstraction people use (protocol layers) is a *functional decomposition* rather than an object-oriented decomposition (e.g., [6]). The approach we have taken is to treat protocol layers as *processing elements* which are encapsulated by objects (see section 2.6).

In addition, we have broken out some functional objects which encapsulate various optional algorithms that can be used to implement different policies. Examples of such *function objects* are flow-control objects and policy-based packet filtering objects.

2.4 External Representation Issues

Packets pose external representation problems which are similar to those encountered in dealing with persistent objects: they enter the system as a bit-string and must be converted to an object to be useful. Since packets are much simpler than arbitrary objects, we use a solution which takes advantage of the packet's structure rather than use a general purpose *persistent object* solution. In particular, packets don't normally contain references to other packets, so we need only worry about the representation of simple scalar data types and composite structures formed from them.

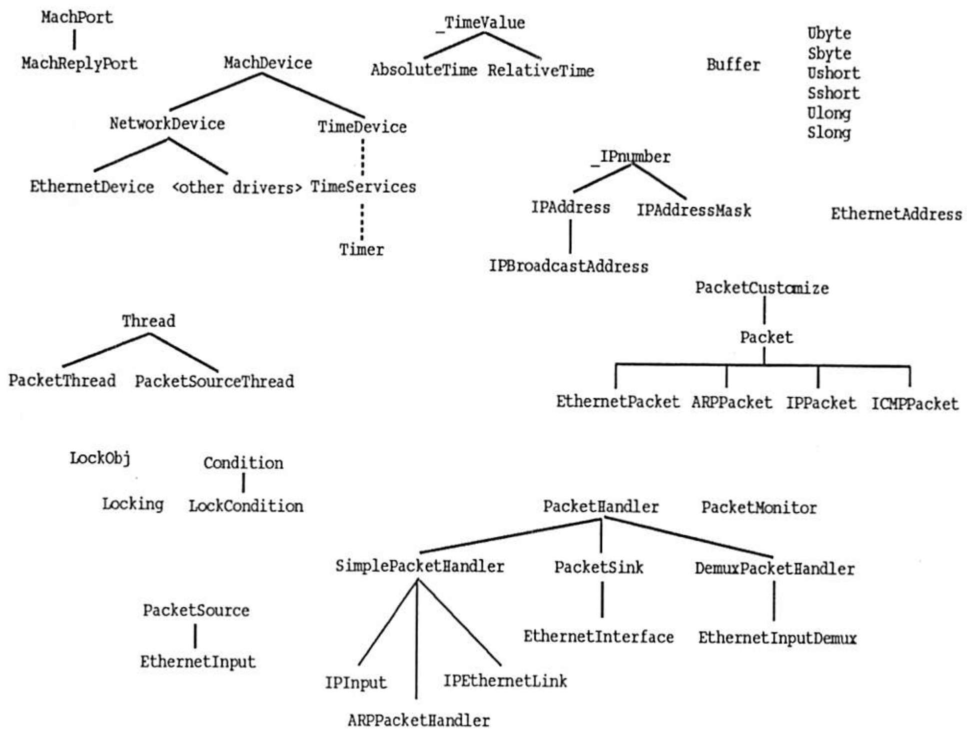

Figure 1: Some Major MTG Toolkit Classes

We handled the external representation problem by defining a set of special scalar data types which store data internally in machine-dependent form. Each of these data types has input and output operations defined on the `Buffer` class. Instances are used to cache in *machine* format data elements stored in *network* format in the message. The definitions of these data types and their input/output operations are all that need to be redefined to handle data representation problems on a new hardware platform.

In addition, robustness in packet-switches requires stringent error-checking, which can be quite complex in the case of some types of packet headers. Such code is relatively easy to provide as part of the process of converting the network representation of a packet into a convenient internal form (see section 3.2).

2.5 Dynamic Polymorphism

Packets have multiple layers (e.g., consider a `IGPPacket` encapsulated in an `IPPacket` encapsulated in an `EthernetPacket`). Each layer provides different information and supports different functionality, so we need a different data type for the packet at each layer. Since only one layer at a time typically is examining the packet, this is an example of an object which changes type as time passes (*dynamic polymorphism*).

Our solution to this problem comes in two parts. First, all packet types are derived from the base class `Packet`, so packet flow can be handled in terms of the base class. Second, we attach a new `Packet`-derived object to the message every time we change its effective type. This object also serves to store parsed header fields in an efficient machine-dependent representation, as well as to supply type-specific functions.

This approach does, however, have some drawbacks:

- Since packets are handled as base class objects, there is little type-checking help from the compiler — all packet flows look equally valid.
- Use of base-class pointers brings with it the *cast-down* problem. Most packet handlers assume any input packet is of a specific type, and cast the `Packet` pointer down to the appropriate subclass.
- The object output by a processing stage may not be the same object passed to it as input. This complicates the code slightly.

We mitigate the first two problems by storing a type code in every `Packet` object, which can be checked at runtime. The third problem can be handled by using a form of delegation — using a *carrier* object of a fixed type, and having it point to an object representing the current header. We rejected this approach because it added a least as much complexity in accessing type-specific functions as it removed from the packet flow code. Such an approach would have meant that each use of a type-specific packet function would require an additional indirection. Since these function calls outnumber simple packet flow operations by a wide margin, we elected the approach which simplifies type-specific functions.

2.6 What Are Objects?

Again we found ourselves in agreement with the approach outlined in [3] in which there are four *types* of objects:

- Data structure encapsulations. These are what people typically think of as objects and we found them useful for classes such as `Packet` where there is data that must be parsed, manipulated, written, etc.
- Process encapsulations. These objects encapsulate processes and accept an object as input and produce a new object as output. This corresponds nicely to classes such as `PacketHandler` which transform packets (e.g., turn a `EthernetPacket` into an `IPPacket`).
- Device encapsulations. These objects provide an abstract interface for controlling devices. In the MTG toolkit this corresponds naturally to entities such as `EthernetDevice`.
- Interface encapsulations. These objects provide interface and conversion procedures between other classes. The MTG toolkit uses this kind of object to provide machine independence by converting between machine-independent network-order bitstrings (e.g., `Buffer`) and higher level objects containing efficient machine-dependent representation of header field values (in `Packet`-derived classes). Our approach uses a group of special integer types which hold network-derived values, combined with a set of "stream"-like I/O operators on the `Buffer` class.

2.7 Operating System Issues

Many packet switches are not built on top of an operating system: efficiency considerations lead to building them directly on top of the system hardware. We have chosen to implement the MTG on top an of OS to increase portability and to facilitate ease of debugging and experimentation. We are currently building the toolkit using the Mach 3.0 micro-kernel facilities [2], though the development environment makes use of Mach's Unix-compatibility.

To increase portability, we have designed the MTG around a set of OS-interface classes that provide the minimal services needed by packet switches from their environment: threads of control, timing services, and I/O. By reimplementing a few classes, the toolkit can be ported to a different operating system, or even to bare hardware.

2.8 Support for Parallelism

Because experimenter efficiency is our *primary* goal, we are willing to trade-off some implementation efficiency for flexibility, code reusability, etc. To support high-performance switching, we are building the toolkit so that it can take advantage of multiple CPUs: more hardware can be added to improve performance.

Packet switches are characterized by a high level of potential parallelism, since packets have minimal interaction with each other during the switching process. This leads to the notion of giving each packet its own private thread of control, eliminating the need to lock the frequently-accessed packet header. The policy for attaching threads of control to packets is provided by classes derived from `PacketSource`, described in section 3.4.

Some control structures, such as routing tables, are shared between multiple packets or packets and other control flows. These structures typically need locks; unfortunately there is no language support available to express this fact. Classes that manage such shared structures have to provide appropriate locks in the class objects, and use a correct locking protocol in the access functions. We routinely make all data either private or protected, so correct locking can be enforced by appropriately written access functions.

There are some subtleties involved in the design and implementation of parallel packet switches. We are drawing on our experiences in related projects ([4]) in designing the toolkit for parallel operation.

2.9 Wiring/Initialization

A toolkit isn't useful if it's difficult to put the pieces together. We considered and rejected several toolkit designs which we felt complicated the process of connecting pieces together (wiring) and initializing these pieces. One approach we considered involved a "pure" object-oriented approach that put all the processing code into subclasses of `Packet`. This approach exacerbated the dynamic polymorphism problem and complicated the process of modularizing packet handling functionality.

Another approach (similar to that used in the X-kernel [5]) would have treated the system as a series of protocol layer objects, each with a thread which would read input from a queue, perform some processing, and then write to an output queue. This approach imposed significant constraints on the code written by toolkit users, and would have interfered with experiments involving modifications to the system structure (which may be as interesting as algorithmic experiments).

We finally settled on a design similar to that used by Zweig and Johnson in their Conduit abstraction [8]: generic packet objects flow along a connected set of processing stages (typically representing protocol layers). A flow of control is attached to a packet when it arrives, and pushes it through subsequent processing stages.

For example, a thread waits to read a packet from an Ethernet interface. When one arrives, the thread calls the `Process` function of the Ethernet input processing stage with the packet as an argument. After handling the Ethernet packet header, this function calls in turn either the `Process` function of the ARP processor (for an ARP packet), or the IP input stage (for an

Internet protocol packet). When the packet is finally disposed of, the thread returns from all the Process functions and loops back to read another packet from the interface.

Our approach differs from the Conduit model in being finer-grained: we are concerned not just with the layering of protocols but with the implementation of those protocol layers. For example, we view data flow as packets flowing through uni-directional PacketHandlers, rather than bi-directional Conduits since this better models the flow of packets inside a packet switch, where packets are typically independent of one another. Where appropriate, we have built up bi-directional objects from groups of PacketHandlers.

Constructing a gateway then consists of creating a selected set of processing-stage objects and *wiring* them together appropriately. Since all these objects share a common entry point to which packets are passed, any object can potentially pass packets to any other. This simplifies wiring, since no object need know anything more about the next stage than the fact that it's another PacketHandler. Only the configuration code need know the actual types of the objects.

The drawback to this approach is loss of type-safety — there is no guarantee that a packet that reaches a particular stage is of an appropriate type. The Conduit abstraction in [8] attempts to solve this problem by doing type-checking at wiring time. Each Conduit subclass knows which others it is willing to connect to, and information is exchanged at connection time to permit enforcement. Unfortunately, this requires that every Conduit subclass know about every other, which violates our extensibility goal. It also doesn't guarantee type-safety, since there's no guarantee that the packets coming out of a connection are *really* the right type — the problem has only been pushed down a level, into the internals of the processing stage.

Therefore, we chose to ignore type-safety as a wiring consideration. The initialization code is relatively small and straightforward, so we don't think we introduce a major problem in doing this. Also, all of our packets carry a type code, so that each processing stage can check each incoming packet for an appropriate type.

3 Data Path Classes

These classes represent packets and their processing.

3.1 Buffer

The Buffer class implements uninterpreted variable-length Byte arrays, suitable for storing packets in network data format. It provides various access methods to interpret the data in the array as simple structured values, such as 32-bit unsigned integers. It also contains member functions that facilitate inserting and deleting data, such as packet headers.

Each buffer object also has a *cursor* associated with it. We provide "stream"-like << and >> operators between the buffer and various network data types. These operators read or write at the cursor position and advance the cursor by the size of the element transferred. Parsing or composing a message header reduces to declaring a set of variables of the appropriate network data types and reading or writing them in order using the stream operators.

3.2 Packets

The Packet class represents a packet as it has been processed by the MTG, including its data, processing history, and any other per-packet information a gateway implementation wishes to keep. Subclasses of Packet represent the various types of message headers that the system needs to process, and contain member functions that operate on such headers.

As a message flows through the system, it is actually represented by a pointer to a Packet object. This object contains a pointer to the Buffer object containing the message data, and any state information about the message at its current stage of processing. This will typically include access functions for various header fields.

Also attached to the packet object is a doubly-linked list of previous packet objects that represent earlier processing stages of the message. These can be used to carry along header information from stripped headers, or other data of interest to higher-level processing layers (e.g., whether the packet was sent to the broadcast address, or the packet's retransmission status). This chain of processing history is also potentially useful in debugging new network algorithms or network faults.

The Packet class includes a facility for placing a *return marker* in a packet object. A return marker indicates that, instead of deleting the packet after it has been transmitted, its header should be *rolled back* and it should be sent on to a packet handler for additional processing. While this mechanism is quite general, we only support rolling back added headers and not restoring removed ones, since the main use of this facility is to provide a simple means of handling retransmissions and multicast transmission. An integer can be returned as part of this process, which is used to indicate the disposition status of the packet on output (e.g., whether the device interface successfully transmitted the packet). Packet instances do not have to represent actual headers in a message — they can also be used to store state information needed by retransmission algorithms (e.g., number of previous transmission attempts) or return markers.

3.3 PacketHandlers

The PacketHandler class provides a mechanism for encapsulating packet processing stages. It contains a public entry point to which packets can be handed for processing, and a virtual function for passing a packet on to the next appropriate processing stage.

PacketHandlers provide a convenient abstraction which supports the notion of wiring together objects which process packets. Each processing module performs its functions without knowing anything about the next stage except that it's a PacketHandler. This allows a gateway implementor to build a gateway in a "tinker-toy" fashion by plugging together user-developed objects and toolkit-supplied objects subject only to the constraint that they be derived from the PacketHandler class.

In addition to providing a standard packet processing model, PacketHandler provides a uniform, built-in mechanism for instrumenting and debugging experimental gateway designs. The PacketHandler class maintains a list of PacketMonitor objects and allows each one to examine packets before they are processed. PacketMonitors take a read-only Packet object as input, and look at it in some useful way, possibly recording information, or sending signals to a debugger (e.g., implementing a *packet breakpoint*).

3.4 PacketSource

The PacketSource class provides member functions that create or get packets and attach a thread to them. A thread is a "flow-of-control" which pushes packets through the processing stages (PacketHandlers) of the MTG. Packet source objects are used to represent input interfaces, the dequeuing side of queues, the timeout portion of retransmission objects, and time-based packet creators such as routing update generators.

A PacketSource may own one or more threads; when the associated packet is destroyed, enqueued, or transmitted, the thread is released back to the PacketSource. Two options for thread behavior are provided, controlled by a constructor argument. Under the first option the `PacketSource` object is initialized with a fixed number of threads, each of which can push a packet through the appropriate processing. Under the second option, only a single permanent thread is created. When this thread acquires a packet, it creates a new `PacketThread` to process the packet, hands the packet over to it, and goes back to trying to get another packet.

3.5 Queues

A `Queue` is a composite object containing a `PacketHandler` and a `PacketSource`. The `Process` member function of `Queue` terminates any thread of control processing a packet after storing the packet in the queue. The `PacketSource` component of the `Queue` provides its own thread of control which repeatedly calls the `GetNext` member function. Different implementations of `GetNext` can provide FIFO queues, Fair Share queues, etc.

Besides their obvious use as buffers between asynchronous processing stages, queues are often used as a means for implementing packet scheduling policies in packet switches. These policies include flow control, retransmissions, and handling of multi-priority traffic. Our design allows for easy implementation of such policy mechanisms using the virtual function `GetNext` to implement a selection operation over the packets currently stored on the queue.

3.6 Multicasters

One operation that packet switches may perform is *multicasting* – duplicating a packet and forwarding it to multiple destinations. Using the MTG toolkit, this can be done by adding a *multicaster* object to an implementation and wiring it into the proper data path.

A multicaster object can be implemented by subclassing the `PacketHandler` class in one of several ways:

- Use the return mechanism to have the packet sent back to the multicast object after the packet has been transmitted. A multicast header `Packet` object would be added to the front of the packet, which would hold the destination list, an index into the list, and the return marker.
- Use a specialized `SendNext` function that makes copies of the packet and puts them on a queue and either i) generates a new `PacketThread` for each, or ii) uses one `PacketThread` which transmits one packet at a time.
- Queue the multicast packet on a special subclass of `Queue` that fetches it N times before dequeuing it. This would use a `PacketSource` with a specialized `GetNext` function and the addition of a special multicast header to the packet (to carry the list of destination addresses).

4 Control Classes

Most of the effort to date has been expended on the development of the data path classes. So far we have identified the following control classes:

- Link. These objects contain the information about the state of "links" to neighboring switches and encapsulate the link up/down protocol. They notify the routing object of any changes in link quality.

- Routing. This object produces routing updates for distribution to other switches (typically triggered by a change in connectivity or a change in link quality) and processes routing updates from other switches. The result of processing these routing updates is a forwarding table which specifies for any destination, where a packet should be sent.
- Forwarding Table. This object determines the next object to handle a packet based on the packet's source, destination, type-of-service requirements, administrative constraints, etc.
- Distribution. This object is responsible for distributing routing updates. It encapsulates the protocol (typically a flooding algorithm) and maintains state about what neighboring switches need to know.
- Congestion Control. These objects encapsulate the algorithms used to control the rate at which packets are sent to a neighboring switch.

5 Gateway Design Example

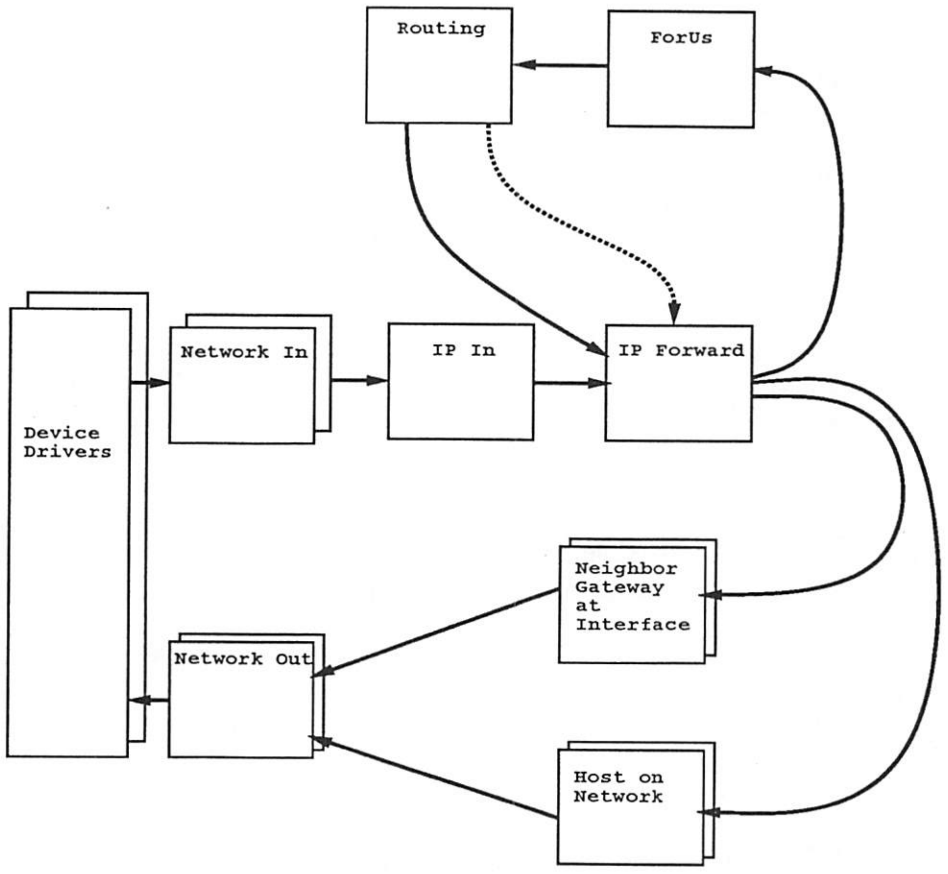

Figure 2: Simple Gateway

A gateway is a packet switch that routes packets between a variety of networks. A group of cooperating gateways forms an internetwork, permitting data flow using a common protocol among the hosts attached to component networks. Figure 2 is a high level view of the components of a single, simple, gateway that handles the Internet [7] protocol. To construct this gateway using the MTG toolkit one needs to do the following:

- Create the IP In, IP Forward, ForUs, and Routing objects. Only one of each is required in this gateway. All these objects are `PacketHandlers`. IP In handles the details of the Internet Protocol headers.

  The IP Forward and Routing objects cooperate to implement the desired routing protocol. IP Forward looks up packet destinations to determine how to forward them, while packets delivered to the Routing object contain routing information updates from other gateways, which are used to update the routing table internal to IP Forward (dashed arrow in Figure 2).

  ForUs is a simple demultiplexing `PacketHandler` that examines the protocol number field of incoming packets addressed to the gateway itself, and delivers the packet to the appropriate object for further processing. Only the Routing object is shown here, but typical gateways implement other protocols for monitoring, control, and network testing (these have been omitted from this example for clarity).

- "Wire" the permanent objects just created. This is typically done via the `Connect` member function provided in the `PacketHandler` class. This function takes a pointer to the next packet handler and may take an additional argument specifying the type of packets it should send to the next packet handler. For example, this argument is used to specify the packets that the ForUs object should send to the Routing object.

  The Routing object is responsible for maintaining the table of destinations contained in IP Forward which specifies how it delivers packets. This connection is not based on the flow of packets and is indicated by a dashed line in Figure 2.

- For each network interface, create a Device Driver object (which encapsulates the hardware interface), a Network In object (which parses local net headers and delivers them to the next protocol layer), and a Network Out object (which adds local net headers to outgoing packets). The Device Driver object (an instance of the NetworkDevice interface class) is used by the Network In and Network Out objects to read and write message buffers. Network In is a `PacketSource` object (it is blocked from reading packets until wiring is complete); Network Out is a `PacketSink` — a variant of `PacketHandler` that deletes a packet after processing it rather than delivering it to another `PacketHandler`. In actual practice, we have created composite objects that package the Device Driver, Network In, and Network Out objects into a single higher-level object, further simplifying the wiring process.

  In addition, a "Host on Network" object is created to handle packets destined to hosts on the attached network. This object and the Network Out object are passed to a function of the Routing object, which stores them in its database (and wires the Host on Network object to the IP Forward object). The IP In object is wired to the output of Network In using a connect function specifying that IP protocol messages be delivered to it.

- Once all interfaces are wired, some final configuration parameters must be passed to the Routing object. These name one or more of the gateways which have networks in common with this one. This information is passed to the Routing object, which creates a Neighbor-Gateway object for each. Delivering a packet to this object causes the packet to be sent to the actual neighboring gateway for further forwarding.

The Routing object enters the NeighborGateway object in IP Forward's table with appropriate routing information attached (the nature of this depends on the routing algorithm). The NeighborGateway object is also wired to the appropriate interface over which traffic must be sent to the neighbor.

- Now that initial wiring is complete, each Network In object has its `Initialized` function called. This releases the threads contained therein to start reading and processing incoming packets.

- Routing updates reflect internetwork topology changes. Updates received by the Routing object are used to control the creation and destruction of NeighborGateway objects as well as to update the entries in the IP Forward table.

For a simple fixed-routing demo gateway, the configuration code for the steps described above is less than a page long; we expect only a small increase in size for more complex gateway architectures.

To change routing protocols in this gateway, one needs only to create new subclasses of `PacketHandler` to replace Routing and possibly IP Forward. Additional interface types can be added by creating new subclasses for Device Driver, Network In, and Network Out. In a similar manner, the IP protocol suite can be replaced, or a different protocol suite run in parallel with it, by having other `PacketHandlers` connect to the network level using an appropriate high-level protocol identifier.

6 Language Issues

We decided to use C++ for the obvious reasons of modularity, code reuse, etc. However we have encountered the following issues:

- The Mach OS provides multiple address spaces (tasks) which we would like to use to implement packet-switch security features and to support "firewalling" (i.e., segregating different parts of the packet-switch code to localize the effects of errors).

  Multiple address spaces cause trouble in a language such as C++ where pointers are prevalent, especially when not all pointers are generated explicitly (e.g., compiler-created vtable pointers). Moving objects across address-space boundaries requires converting them to some sort of external representation. Accessing data in an object in a different address space requires either i) that objects recognize where task boundaries are and directly deal with external representation issues, or ii) the use of *proxy* objects, which hide the task boundaries and external representation issues but which must be generated by hand and which complicate the wiring problem.

- Dynamic polymorphism (cast-down problems). Packets change as they move through a switch and headers are parsed, stripped and built. `PacketHandlers` receive their argument as a pointer to a generic `Packet` object. While many of the operations on a packet are virtual, each type of packet also may support some non-generic operations. To access these member functions requires that a `PacketHandler` access a `Packet` as a particular derived class: i.e. the packet pointer must be *cast-down* to a more specific type of packet. Language support for this capability would have allowed us to deal with this problem without abandoning type safety.

- Multiple Inheritance. We use multiple inheritance sparingly in our classes. Partly this is due to the tools we use, which seem to have problems with programs using this feature, but it also reflects a design decision we made early in the project. We avoid multiple inheritance where possible, since it seems to make code more difficult to understand and complicates debugging.

- Overloading. We make relatively little use of operator overloading. One reason for this is that data structures used in packet-switching have little in common with integers, and the meaning of most C++ operators on such data structures wouldn't be intuitive. Instead, we prefer to use named functions. There are a few exceptions: we overloaded the subscripting operator in a number of array-like classes, the assignment operator was overloaded for certain complex data structures, and some integer-like classes had certain arithmetic operators defined on them. In general, we used function overloading sparingly (constructors, buffer operations, connect operations, etc.) as there seemed to be few occasions where it enhanced comprehensibility.

- Default arguments. We used default arguments to good effect in a number of places, preferring this to function overloading. However, the restrictions on defaulting only the last arguments in the list forced us to order some function argument lists in an unnatural order. This can be solved by adding overloaded functions, but this is cumbersome to the programmer. A more flexible argument defaulting mechanism, like that of Ada, would have been useful on occasion.

- Exception Handling. This is not currently supported in C++ but clearly necessary for real systems. We have had prior experience with exception handling in packet switches [4] and have found it very useful in improving the robustness and debuggability of the resulting systems. Currently our code is littered with comments flagging places where we want to generate exceptions as soon as the language supports them.

- Use of Templates. Templates would simplify *some* aspects of implementing routing algorithms. (For example, certain routing algorithms compute a *minimum metric* path, but don't care what kind of metric is used (delay, hops, etc.).) Templates appear to be primarily useful in creating *collection* classes like lists. However they don't seem to be of much use in implementing subclasses that include a lot of subclass-dependent boilerplate, like the subclasses of `Packet` do in our toolkit.

- Wrappers. It is sometimes the case that we want to provide some *generic* processing in a base class which is augmented (not replaced) in a subclass. There is no language mechanism which allows the toolkit builder to specify that subclasses should execute a base class member function before (or after) calling the subclass member function (e.g., a wrapper feature [1]). This functionality would have been useful in developing base classes that represent shared objects: the base class member function could lock and unlock the lock "around" the access function defined in the subclass.

- Automatic Calling of Conversion Operators. As a convenience we want to allow the toolkit user to initialize IP addresses from an integer and convert it back to an integer (say for hashing). In general, however, it is an error to add an `IPAddress` and an integer. Unfortunately, providing an `IPAddress` constructor with an integer argument and a conversion to `int` means that these errors will not be caught.

6.1 Overall Impressions of C++

We studied a number of candidate languages before starting the coding phase of our project. We chose C++ for several reasons: it was widely-supported, giving us portability to many platforms; it was reasonably efficient (always a concern in packet-switching); and it supported the object-oriented programming model we wanted to explore in the toolkit.

While C++ wasn't perhaps ideally suited to our needs, especially in the area of dynamic polymorphism, our experience with the language has been generally positive. Most of our difficulties with the language have been minor, or are being addressed as C++ matures. Other problems are common to most present languages (e.g., multiprocessing issues). Some may be addressed by new techniques as experience with the language grows.

7 Conclusions

The MTG toolkit is designed to support a variety of experiments in gateway design on a wide range of platforms. We are currently engaged in testing the toolkit concepts by building some simple gateways. We have implemented a simple static-routing gateway and a EGP stub gateway using the toolkit. We plan to implement advanced gateway algorithms and to use this implementation experience to improve the toolkit architecture and classes. We plan to make this toolkit available to experimenters later this year.

References

[1] *Symbolics Common Lisp – Language Concepts.* Symbolics, Inc., August 1986.

[2] R. Baron, D. Black, W. Bolosky, J. Chew, R. Draves, D. Golub, R. Rashid, A. Tevanian, and M. Young. *Mach Kernel Interface Manual.* Carnegie-Mellon University, August 1990.

[3] J. M. Coggins. Designing C++ libraries. In *1990 Usenix C++ Conference*, pages 25–36, The USENIX Association, April 1990.

[4] W. Edmond, S. Blumenthal, A. Echenique, S. Storch, T. Calderwood, and T. Rees. The butterfly satellite imp for the wideband packet satellite network. In *Proceedings of ACM SIGCOMM*, pages 194–203, August 1986.

[5] N. C. Hutchinson and L. L. Peterson. Design of the X-Kernel. In *Proceedings of the ACM SIGCOMM '88 Symposium*, August 1988.

[6] B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.

[7] J. B. Postel. *Internet Protocol.* RFC 791, ISI, September 1981.

[8] J. Zweig and R. Johnson. The conduit: a communication abstraction in C++. In *USENIX C++ Conference Proceedings*, pages 191–204, April 1990.

An AWK to C++ Translator

Brian W. Kernighan

AT&T Bell Laboratories
Murray Hill, New Jersey 07974
`bwk@research.att.com`

ABSTRACT

This paper describes an experiment to produce an AWK to C++ translator and an AWK class definition and supporting library. The intent is to generate efficient and readable C++, so that the generated code will run faster than interpreted AWK and can also be modified and extended where an AWK program could not be. The AWK class and library can also be used independently of the translator to access AWK facilities from C++ programs.

1. Introduction

An AWK program [1] is a sequence of pattern-action statements:

```
pattern     { action }
pattern     { action }
...
```

A *pattern* is a regular expression, numeric expression, string expression, or combination of these; an *action* is executable code similar to C. The operation of an AWK program is

```
for each input line
        for each pattern
                if the pattern matches the input line
                        do the action
```

Variables in an AWK program contain string or numeric values or both according to context; there are built-in variables for useful values such as the input filename and record number. Operators work on strings or numbers, coercing the types of their operands as necessary. Arrays have arbitrary subscripts (''associative arrays''). Input is read and input lines are split into fields, called `$1`, `$2`, etc. Control flow statements are like those of C: `if-else`, `while`, `for`, `do`. There are user-defined and built-in functions for arithmetic, string, regular expression pattern-matching, and input and output to/from arbitrary files and processes.

The standard implementation of AWK is an interpreter: an AWK program is parsed into an internal representation, which is then interpreted by a set of routines.

AWK is a closed language; there is no access to libraries or to separately written code. This often forces users into contortions to do some operation that would be much more naturally expressed in some other language. A translator into C++ provides a way to extend AWK programs, by combining them with C++ code.

AWK is in some ways a seductive tool (see, for example, [2, 3, 4]) and there are numerous examples of AWK programs that grew from sensible small versions into awkward large ones. A translator provides an escape path: a program may be translated so that further development takes place in a more suitable language.

Both of these uses require that generated code be especially readable and easy to work with. The primary goal throughout the experiment has been that the translated output should be as close as possible to the original AWK input. This means that we want to define variables that have the semantics of AWK variables and use them in the natural syntax in expressions with the usual C operators, C and AWK functions, and C built-in data types. AWK syntax is close enough to C to make an exact match feasible in many places, and provide a reasonable mapping in others.

It is necessary to duplicate AWK semantics, but that is not sufficient. There is already an excellent AWK to C converter, that generates C that exactly matches the semantics of AWK. Roughly speaking, an program will run about twice as fast as the corresponding AWK program, so if one wishes only to speed up an existing AWK program, is quite satisfactory. The output of however, was never meant for human consumption, and thus it is not appropriate for augmenting or extending AWK code.

This project has several components. Their development was carried on in parallel, since they are related and activities in one area affect the other areas. I have tried to separate these as much as possible but there is still room for confusion. Here is a sketch of the pieces and events.

The *AWK interpreter* is a C program originally written in 1977 and much modified since then. For most people, the interpreter *is* AWK. The first step was to translate the interpreter into the C subset of C++ and then to make some minor changes in implementation to use C++ better. This version of the interpreter could (but does not) replace the standard C version. There was also no need to make this a C++ program but it was good practice.

The second step was to modify the interpreter so that instead of interpreting AWK programs, it translates them into C++; this program is called the *translator* and the C++ it produces is called *generated code.*

The generated code does not stand alone; it assumes that it will be compiled (by a C++ compiler) with a *header file* that contains class declarations for AWK data types and loaded with a *library* of separately compiled functions for input-output, field splitting, regular-expression matching, etc. Thus the third step is the development of this header file and library. These are written in C++.

The translator, header file, and library are interdependent, since each performs actions or provides services that the others depend on. There are often trade-offs among them, since one can do more work in one place to simplify life in another. Among the issues that can be traded off are complexity, efficiency, and readability of generated code.

2. Translation

The translator parses an AWK program creates a parse tree and walks it recursively; at each node it calls a routine that generates C++ code. The translator makes only relatively simple use of C++ facilities; its origin as a C program is very clear.

The generated code is meant to be compiled with a header file `Awk.h`, to be discussed in the next section, and loaded with a library of run-time routines. The generated code is based on this template:

```
#include "Awk.h"

// declarations of user variables and functions, if any

main()
{
        BEGIN();          // if there is a BEGIN block
        while (getline() > 0) {
                // code for pattern-action statement 1
                // code for pattern-action statement 2
                // ...
        }
        END();            // if there is an END block
}

// user function definitions, if any
```

Considerable effort has been devoted to generating readable code. For example, no redundant parentheses are produced for expressions, and no redundant braces are produced around non-compound statements. The output is piped through the C beautifier `cb` so it is properly indented. Declarations of external variables are sorted so that variables are easy to find in large programs. Default arguments and multiple declarations are used for functions like `substr` and `split` that can be called with different numbers of arguments.

There are some AWK notations that simply cannot be made to look the same in C++. For example, there is no C exponentiation operator, so `x^y` becomes `pow(x,y)`. (The alternative, overloading the C ^

exclusive operator, must be rejected because its precedence does not match the AWK precedence for exponentiation.) There is no explicit operator for concatenation in AWK; the translator generates a function call instead of overloading some operator, because there is no suitably mnemonic C operator with the right precedence. The common cases of concatenation of two or three strings are handled with `cat(s1,s2)` and `cat(s1,s2,s3)`; longer concatenations use nested function calls. Constant regular expressions must be delimited by quotes instead of slashes, and an extra layer of backslashes must be added to protect embedded backslashes.

The notation for fields is a problem, since there is no way to use '`$`' as AWK does. After some experimentation, I decided to use the variable `F` so that fields are called `F(0)`, `F(i+1)`, and so on. There are also definitions for `F0`, `F1`, etc., so that an expression like `$1 > $2` can be expressed as `F1 > F2`.

A small example will illustrate many of these points. This AWK program reads a list of numbers and prints the list with serial numbers and percentage of the total:

```
    { x[NR] = $1; sum += $1 }

END { if (sum != 0)
            for (i = 1; i <= NR; i++)
                printf("%2d %10.2f %5.1f\n", i, x[i], 100*x[i]/sum)
    }
```

The translator generates this code:

```
#include <stdio.h>
#include "Awk.h"

int     i;
double  sum;
Array   x;
void    END();

main(int argc, char *argv[])
{
    Awkinit(argc, argv);
    while (getline() > 0) {
        x[NR] = F1;
        sum += F1;
    }
    END();
}

void END()
{
    if (sum != 0)
        for (i = 1; i <= NR; i++)
            printf("%2d %10.2f %5.1f\n", i, (double) x[i], 100 * x[i] / sum);
}
```

The basic type is an `Awk`, which captures the semantics of AWK variables mentioned in Section 1: a string value, a numeric value, or both, depending on usage. In general, most variables in an AWK program would be translated into `Awk`s.

Since it is more efficient to use built-in types, however, the translator attempts to infer the simplest type that will serve for each variable. For example, since the variable `i` is used only as the index of a loop, it can be an `int`, while `sum` is a `double`, and `x` is an `Array`, a type that captures the notion of an AWK array, i.e., an indexable collection of `Awk`s. The coercion `(double)` in the `printf` is necessary to convert the array element, of type `Awk`, to a number for printing. There is no need for a coercion for `i`, however, because its type already matches; in this case, type inference leads to more readable code and potentially more efficient code.

Type inference in the translator is fairly *ad hoc*. A type is associated with each constant. Types are combined at operators to produce a result type; for instance, the result of an addition is an `int` if both operands are integer; otherwise it is `double`. Relational operators always produce `int` regardless of the type of their operands. Types of variables are set by assignment statements and also by usage in expressions and function arguments. The operands of an operator are assigned a tentative type based on the operator; for example, the expression `x+y` implies that `x` and `y` are used arithmetically. This may later be changed to `int` because the variables are only used in `int` contexts. If a numeric variable is used in an explicitly `double` context, such as `sqrt(x)`, or if no further information appears, it will become `double`. Array elements are always assumed to be of type `Awk`, as are fields, since it is too uncertain to propagate an assumed type.

Ideally, one should do a data flow analysis to propagate type information, but instead several passes are made over the parse tree; this way, information that ''flows backwards'' can be handled, so long as the backward path is not too long. For example, in a sequence like

```
{ i = j; j = 2 * k; k = 1; x = y; y = z + 1; z /= 2 }
```

after two passes over the tree it will be concluded that `i`, `j`, and `k` are all of type `int`, while `x`, `y`, and `z` are all `double`.

3. The Awk Class

It is easy enough and quite satisfying to generate clean, clear C++ code, with all the operators in place, and no redundant parentheses or braces. It turns out to be harder to define a class that captures the behavior of AWK variables so that the clean expressions produce the expected results. In this section we will describe the `Awk` class, which is defined in the header file `Awk.h`.

The `Awk` data type keeps track of the value and state of a variable:

```
class Awk {
  private:
        double  fval;    // floating-point value if currently valid
        String  sval;    // string value if currently valid
        int     state;   // which values are currently valid
        ...
```

(The fragments of `Awk.h` presented here have been somewhat simplified to show the essence without bogging down in details.) The `state` variable holds only two bits, which are set if the numeric or string values or both are currently valid. `String` provides reference-counted strings; it is a tiny subset of the standard C++ library string package.

The next step in the class definition is constructors to create `Awk`s:

```
  public:
        Awk() : sval("")          { fval = 0.0; state = STR|NUM; }
        Awk(int i)                { fval = i; state = NUM; }
        Awk(double f)             { fval = f; state = NUM; }
        Awk(cchar *s) : sval(s)   { state = STR; }
        Awk(Awk &a)               { if (a.state & STR) sval = a.sval;
                                    fval = a.fval; state = a.state; }
```

The type `cchar` is an abbreviation for `const char` here and in the sequel.

Similar functions are necessary for assignment of values to `Awk`s:

```
        Awk &operator =(int i)        { fval = i; state = NUM; return *this; }
        Awk &operator =(double f)     { fval = f; state = NUM; return *this; }
        Awk &operator =(cchar *s)     { sval = s; state = STR; return *this; }
        Awk &operator =(Awk &a)       { if (a.state & STR) sval = a.sval;
                                        fval = a.fval; state = a.state;
                                        return *this; }
```

and for increment operators like += and for ++ and −−.

There are also ''conversion functions'' for fetching the numeric and string values of `Awk` variables:

```
operator double()
        { return state&NUM ? fval : (state |= NUM, fval = atof(sval)); }
operator cchar *()
        { return state&STR ? sval : (state |= STR, sval = ftoa(fval)); }
```

These are used implicitly when calling a normal C function that expects one of these types as an argument. For example, since `sqrt` expects an argument of type `double`,

```
sqrt(awkvar)
```

is really

```
sqrt( (double) awkvar )
```

The alternative of requiring explicit casts for ''downward'' conversions from `Awk` to built-in types is unacceptable because it severely affects readability; even a few casts are undesirable.

The real complications begin with the arithmetic and relational operators. In an AWK arithmetic expression involving the operator +, there are five possible combinations:

```
Awk + Awk
Awk + int
int + Awk
Awk + double
double + Awk
```

Each of these produces a `double` value.

The obvious way to handle this is to overload the + operator as a `friend` function:

```
friend double operator +(Awk &, Awk &);
```

The arguments must be passed by reference since the function has to be able to cause the side-effect of updating the numeric state of each argument if necessary. A `friend` function is required so that the left-hand operand can be an `int` or `double`; if a member function were used, the left-hand operand would have to be an `Awk` and this would preclude expressions like `1+Awk`.

Unfortunately, the simple solution doesn't work, because an expression like

```
Awk + int
```

is ambiguous; it could be parsed as either of

```
Awk + (Awk) int
(int) Awk + int
```

The problem is that basic types can be promoted ''up'' into `Awk`s and `Awk`s can be converted ''down'' into basic types, and there is no way to state which choice is preferred. C++ provides an elaborate sequence of rules that determines how type-matching of functions is done, but when it is finished, if there are two matched functions, the construction is ambiguous.

The problem is that in AWK all possible conversions are legal; given both upward and downward implicit conversions, the only way to capture this at compile time is to spell out all possible combinations:

```
friend double operator +(Awk &, Awk &);
friend double operator +(Awk &, int);
friend double operator +(Awk &, double);
friend double operator +(int, Awk &);
friend double operator +(double, Awk &);
// and so on for - * / %
```

All told there are 5×5 functions for arithmetic operations.

As it is for arithmetic operators, so it is for relationals, except that there are more combinations and the semantics imposed by AWK are more complicated: it is necessary to look at the state of each variable in a comparison to determine whether the comparison is numeric or string. This makes another 42 functions (6 operators, 7 type combinations). Fortunately, most of these are trivial and can be expanded in-line.

By the way, it is necessary to distinguish `int` from `double`, rather than relying on the automatic coercion that would otherwise take place. Consider the expression

```
Awkvar == 0
```

In the absence of explicit functions for `int`s, 0 can be a `double` or a `char*`, so this construction would be ambiguous.

4. Fields

The next complication is the treatment of fields. Fields in AWK are for the most part the same as ordinary variables except that they have potential side effects, and there are significant efficiency considerations since field-splitting is expensive.

Each time a new line is read, the input record `$0` is set, but it is undesirable to set `$1`, etc., until they are actually needed. In addition, if any field is assigned to, that invalidates the value of `$0`, but it is undesirable to recreate `$0` until its value is needed again. Similarly, if `$0` is explicitly assigned to, that invalidates `$1`, etc. Thus some form of lazy evaluation is called for.

To make all of this work, it is necessary to intercept every reference to any field. In the interpreter, there is type information in each variable that indicates whether the variable is a field. That requires a run-time test for each access to any variable, so it seems better in the generated C++ code to implement fields as a separate type, thus moving the test to compile-time.

Thus fields are implemented as a new type, called a `Field`. The initial try was to derive `Field` from `Awk`, so that most operations would be inherited, but this doesn't work because operations performed on derived objects may bypass the explicit assignment and conversion operations in favor of implicit ones and thus avoid the code meant to trap references to fields.

A `Field` really isn't an `Awk`, since it may never be used as a plain `Awk` without taking the side effects into account. Thus a `Field` *contains* an `Awk`:

```
class Field {    // an individual field
  private:
        Awk a;
        void rvalue(), lvalue();
        operator double() { rvalue(); return (double) a; }
  public:
        friend double operator +(Field &x, int d) { return (double) x + d; }
        Field &operator =(Field &x) { x.rvalue(); lvalue(); ... }
        // ...
};
```

Within this class definition, in every context where the value of the field will be used, a function `Field::rvalue` is called to ensure that any necessary field-building is performed. In any context where a field will be assigned to, a function `Field::lvalue` is called to build fields and to record any information about invalid state. The `lvalue` function must also be called from a few functions such as `sub` and `gsub` that can alter fields implicitly.

With one exception, there are no explicit variables of type `Field`; rather, fields are members of an array managed by a class called `Fields`:

```
class Fields {   // manages an array of Field's
  private:
        Field fields[100];
  public:
        Field &operator()(int n) { return fields[n]); }
};

Fields F;          // the fields are stored here
```

The only `Fields` operator is `()`, which is used to access an individual `Field`. Thus constructions like `F(0)`, `F(i+j)`, and so on return a reference to the corresponding `Field`.

Since a `Field` does not inherit from an `Awk`, and since a `Field` can be converted to an `Awk` and vice versa, it is again necessary to provide overloaded operators for all possible combinations of `Fields` with other types, and to add some operators to class `Awk` for combinations of `Awk` and `Field`.

It is also necessary to deal with the AWK built-in variable `NF`, which records the current number of fields. If `NF` is referred to, the fields must be computed, or at least counted. The easiest way to do this turned out to be to make `NF` a variable of type `Field` and to add a bit of special code in the `lvalue` and `rvalue` routines to handle it correctly. (The thought of adding another type and another 75 functions to manage it was more than I could bear.)

5. Arrays

AWK arrays are implemented as a separate class `Array`. Thus `Awk v[10]` is not an AWK array in the traditional sense, but `Array v` is. The specific implementation of an array doesn't matter here; the only visible operations are subscripting to implement associative arrays, membership test, and element deletion:

```
class Array {
  private:
        // standard hash table here
        Awk &lookup(cchar *);    // install if not found
  public:
        Array();

        Awk &operator [](Awk &s)          { return lookup(s); }
        Awk &operator [](Field &s)        { return lookup(s); }
        Awk &operator [](cchar *s)        { return lookup(s); }
        Awk &operator [](double f)        { return lookup(ftoa(f)); }
        Awk &operator [](int i)           { return lookup(ftoa(i)); }
        // etc.
};
```

Again, an explicit rule is needed for `int`; otherwise, `x[0]` is ambiguous since `0` is a `double` and a `char*`.

6. Print Statements

The treatment of the AWK `print` and `printf` statements present some interesting problems. In AWK, one writes

```
print e1, e2, e3          # print values of 3 expressions
print e4 > e5             # print e4, redirecting into file e5
print e6, e7 | e8         # print e6 and e7, piping output into e8
```

The es are arbitrary expressions. The goal is to generate code that looks as much as possible like this, something that is natural for human readers and reasonably efficient. The translation has to be legal C++ and provide AWK semantics when executed: values are separated by the value of the variable `OFS` (usually a space) and an `ORS` (usually a newline) is added at the end.

One solution is to define a print function that takes a fixed (large) number of arguments of a single type, a ''print arg''. Constructors are defined that make a print arg from each kind of object that will be printed. There is a default value that marks the end of the real arguments. Default arguments are used to convert calls into the full-length list; short versions are provided for common cases.

```
class Prarg {
        int t;
        union {
                cchar   *sval;
                int     ival;
                double  dval;
        };
  public:
        Prarg()                    { t = 'v'; ival = 0; }  // void marks the end
        Prarg(int n)               { t = 'i'; ival = n; }
        Prarg(double d)            { t = 'd'; dval = d; }
        Prarg(cchar *s)            { t = 's'; sval = s; }
        Prarg(Awk &a)              { t = 's'; sval = a; }
        Prarg(Field &a)            { a.rvalue(); t = 's'; sval = a; }
};

extern  Prarg   Prarg0;            // will mark end of list

void    print(const Prarg& = Prarg0, const Prarg& = Prarg0,  // etc.
              const Prarg& = Prarg0, const Prarg& = Prarg0);
```

Now print statements look exactly the same as they do in AWK, except that the list of expressions must be parenthesized in all cases and there is an upper limit on the number of arguments.

Redirection is handled by a separate class:

```
class Redir {
        char *buf;
  public:
        Redir(char *s)  { buf = s; }

        friend void operator >(const Redir &r, cchar *f);
        friend void operator |(const Redir &r, cchar *f);
};
```

A function named Fprint, which is analogous to print, creates a string that can be printed by Redir::operator>(Fprint,filename) or operator|(Fprint,filename). This permits translation of an AWK statement like

```
print e1, e2, e3 > e4
```

into

```
Fprint(e1, e2, e3) > e4;
```

What about printf and sprintf? Here, the first argument is scanned for format conversion characters that are used to infer the type of each expression in the list. If the type of the expression doesn't match the conversion character, a cast is generated, avoiding redundant parentheses if possible. So, for example, the AWK statement

```
printf("%s %d %f\n", $1, $2+1, 123.4)
```

generates

```
printf("%s %d %f\n", (cchar *) F1, (int)(F2 + 1), 123.4);
```

The casts are unattractive but there seems to be no better solution.

7. Library

Most of the run-time library comprises either functions necessary to implement the operators, or transliterations of functions from the interpreter for regular-expression matching, input and output, field splitting, and so on. Most of these are much the same, although there are some changes in interfaces. Field splitting is probably the most different, since it now has to interface to the different field-handling implementation described above. The library also includes definitions for built-in variables like NR and a routine Awkinit to set up the ARGC and ARGV variables.

One problem arising with the library echoes a previous problem. Consider defining the AWK built-in function length, which returns the number of characters in the string value of a variable. The obvious implementation is

```
inline int length(const Awk &a) { return strlen(a); }
```

Thus length may be called with any type; a constructor will convert this to an Awk, and operator const char* will be called implicitly to coerce the string value for strlen.

One drawback is that calling a constructor is more costly than might be expected; in fact, even though my constructors are all declared inline, some are not inlined because they are too complex. The alternative implementation is again to write out every possible type explicitly. This obviates the problems of constructors and unimplemented features, but it generalizes poorly to functions that have more than one argument, such as cat or substr.

8. Status

Most of AWK can be handled properly. There are a handful of known bugs and constructions that may never work; some of these are intrinsic to the way that I have made trade-offs.

Type inference creates some problems. Consider the program

```
$1 > $2 { i = NR }
END { print i }
```

This generates

```
int     i;
void    END();

main(int argc, char *argv[])
{
        Awkinit(argc, argv);
        while (getline() > 0) {
                if (F1 > F2)
                        i = NR;
        }
        END();
}

void END()
{
        print(i);
}
```

Notice that the type of `i` has been inferred as `int`. Suppose, however, that `$1` is never greater than `$2`. In the interpreter, the variable `i` will have a null value, so the output will be null. In the compiled code, however, since `i` is an `int`, the output will be a literal `0`. This is an example of a trade-off. Is it better to do type inference and get this one wrong, or not to do it and produce less readable code that runs more slowly?

There are minor problems with name clashes. For example, the standard version of `rand` has different properties from the AWK version, which uses the name `Arand`. There are similar problems with `system`, `sprintf`, C++ keywords, and probably others that I haven't stumbled into yet.

The header file `Awk.h` is 625 lines long and the library is 1275, including comments and some debugging code but excluding regular expression matching. The translator is 3700 lines. For comparison, the interpreter in C is 4900 lines.

Performance is mixed, and it is difficult to decide which comparisons are most representative. On some test cases, the compiled code is significantly faster than the interpreter, while on some others, it is somewhat slower. For example, the prototypical AWK program is to compute a word-frequency count:

```
{ for (i = 1; i <= NF; i++) count[$i]++ }
END { for (i in count) printf("%4d %s\n", count[i], i) }
```

The code generated for this is

```
#include <stdio.h>
#include "Awk.h"

Array   count;
Index   i;
void    END();

main(int argc, char *argv[])
{
        Awkinit(argc, argv);
        while (getline() > 0) {
                for (i = 1; i <= NF; i++)
                        count[F(i)]++;
        }
        END();
}

void END()
{
        For (i, count)
            printf("%4d %s\n", (int) count[i], (char *) i);
}
```

This can also be expressed in C++ with standard libraries:

```
#include <String.h>
#include <Map.h>
#include <stream.h>

Mapdeclare(String,int)
Mapimplement(String,int)

main()
{
        Map(String,int) count;
        String word;

        while (cin >> word)
                count[word]++;
        Mapiter(String,int) p (count);
        while (++p)
                cout << dec(p.value(),4) << " " << p.key() << "\n";
}
```

The following table shows running times for four implementations, on an input file of 320,000 bytes with about 600 distinct words:

Map class	16.9 sec.
AWK interpreter	11.5
AWK-C++	8.1
AWKCC	5.7

On another test, a large program that includes representative AWK statements, the results are less favorable:

AWK interpreter	117 sec.
AWK-C++	71
AWKCC	39

Although it might appear that is uniformly faster, this is not true. Computing values of Ackermann's function up to Ack(3,5) shows quite a different picture:

AWK interpreter	21.1 sec.
AWK-C++	1.8
AWKCC	44.5

This stress test of the function calling mechanism appears to show the AWK-C++ translator in its best light.

9. Observations

The job has turned out to be harder than expected, even allowing for my learning curve and the fact that it has been an oft-interrupted back-burner project. In part this is because it is difficult to match exactly an existing program, warts and all, in a new medium; the task would be far easier if I were free to adjust the problem to the solution, rather than being constrained in all directions.

I spent too much time stumbling around trying to get the overloaded operators right. In retrospect, it is quite trivial, but I kept hoping for some alternative to writing out all possible combinations of operands and operators. This would of course be easier if one needed conversions only in one direction, which is the only situation that textbooks typically mention. It may also be easier with templates, but I have not studied the issue.

There was a similar problem with fields, and a lot of trouble getting the semantics exactly the same as the interpreter. Until the `lvalue` and `rvalue` functions were properly in place (in all places!) this just didn't work.

Reference parameters must be carefully thought through so that one does not incur unnecessary overhead, create unwanted temporaries (current C++ implementations warn of this), or inadvertently modify something that should be untouched. The meaning of `const` for reference arguments is in the hands of the implementer; it is quite possible and sometimes desirable to change the value of a `const` object. For example, updating the string or numeric state of an `Awk` does not change its value as far as the rest of the program is concerned, so this is done even for `const` reference arguments.

One must be careful to respect the levels of abstraction when one is building a data type. Several times I inadvertently used an operation on a type from deep within its implementation; this usually caused an infinite recursion. A typical example is using the assignment operator for a type in some function that is indirectly part of the implementation of that type.

Although most of the bugs I encountered were of my own making, I also uncovered perhaps a dozen bugs in C++ (using a development version of cfront) and a handful of C compiler bugs. In general, these were in areas where I was pushing hard on type matching, operator overloading, and conversion functions, exercising them in unusual ways.

C++ has major advantages. Type checking finds lots of errors early in the game that would be terrible to find in a conventional C program. Type-safe linkage extends this checking to separately compiled routines. Operator and function-name overloading are often a help; they are obviously mandatory for an exercise like this one.

C++ diagnostics are very good, pinpointing errors and often suggesting correct code. Compilation and loading are slow, but not a serious problem, at least on a fast machine.

As others have observed, C++ is not a panacea: one can make many of the old mistakes and some interesting new ones as well. In particular, because the meaning of names and even operators depends so much on context, it is harder to see what is going on in ordinary expressions—more work is required to trace through the meaning of an expression.

Nevertheless, the experience has been positive and instructive. C++ made it possible to undertake a project that would have been infeasible in most other languages.

Acknowledgements

I am deeply indebted to Georges Gonthier, Andy Koenig, Doug McIlroy, Bjarne Stroustrup, and Chris Van Wyk, who spent many hours patiently listening to problems and suggesting ways to attack them. I am also grateful to Stan Lippman for bug fixes and enhancements, and to Jonathan Shopiro for some significantly more efficient code. Jon Bentley, Doug McIlroy, Peter Nelson, Rob Pike, Jonathan Shopiro, Bjarne Stroustrup, and Chris Van Wyk made helpful comments on this paper.

References

1. A. V. Aho, B. W. Kernighan, and P. J. Weinberger, *The AWK Programming Language,* Addison-Wesley (1988).
2. J. L. Bentley, *Programming Pearls,* Addison-Wesley, Reading, Mass. (1986).
3. H. Spencer, ''AWK as a Major Systems Programming Language,'' *USENIX Winter '91 Proceedings* (January, 1991).
4. C. J. Van Wyk, ''AWK as Glue for Programs,'' *Software — Practice and Experience* **16**(4), pp. 369-388 (1986).
5. J. C. Ramming, *AWKCC: An AWK-to-C Translator,* AT&T Bell Laboratories internal memorandum (1988).

A Class Library for Solving Simultaneous Equations

Christopher J. Van Wyk

Department of Mathematics and Computer Science
Drew University
Madison, New Jersey 07940

and

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Using a small class library, one can overload the arithmetic operators so that equations can be expressed in C++ programs in a natural way. Once the equations are represented in a program, one can write functions to solve them. This paper describes a small library that illustrates this idea, and comments on the implementation.

Introduction

The `expr` library defines class `Expr`. Objects of class `Expr` can be constrained by a system of simultaneous equations, which the class solves automatically insofar as it can.

For example, the following is a complete C++ program that poses and solves a set of five simultaneous linear equations:

```
#include "expr.h"

void aux(Expr a, Expr b, Expr c, Expr d, Expr e)
{
        a + b + c + d + e == 71;
        20*a + 11*b + 5*c + d + 13*e == 599;
        16*a + b + 15*c + 13*d + 15*e == 771;
        3*a + b + 14*c + 4*d + 9*e == 381;
        a + 3*b + 15*c + 18*d + 14*e == 841;
}

main()
{
        Expr a[5];
        aux(a[0], a[1], a[2], a[3], a[4]);
        for (int i = 0; i < 5; i++) {
                printf("a[%d] = ", i);
                a[i].eval().print();
                printf("\n");
        }
}
```

It produces this output:

```
a[0] = 4
a[1] = 18
a[2] = 5
a[3] = 23
a[4] = 21
```

The C++ code for the `expr` library is well under 500 lines long. It makes extensive use of operator overloading and virtual functions.

An Equation Solver

The class library uses the same algorithm to solve simultaneous equations that is built into `ideal` [Van Wyk 1982] . A brief description here of the solver will help to describe what the class does and how it works.

Every variable in the system has a *dependency representation*. All variables start out *independent*. As equations are solved, variables become *dependent*: they are represented as linear combinations of independent variables. Once a variable's dependency representation contains no variables, the variable is *known*: its dependency representation is a constant.

To process an equation, the solver substitutes the dependency representation for each of the variables in the equation. If the resulting equation is linear, choose one of the variables in it to become *dependent*: henceforth its dependency representation will be a linear combination of independent variables. If the resulting equation is not linear, enqueue the equation to be tried again later.

If the system of simultaneous equations is linear, this algorithm amounts to a version of Gauss-Jordan elimination. If the system is not linear, but a sequence of substitutions can bring each equation into linear form, then the algorithm will still discover the answer.

In the C++ implementation described in this paper, linear equations are processed immediately, while nonlinear equations are placed on a queue `NonLinears`. Therefore, a linear system will be solved automatically (as shown in the first example); the user must ask explicitly, however, that a nonlinear system be solved:

```
Expr x[5];
x[1]*x[2]*x[3]*x[4] == 24;
x[1]*x[2]*x[3] == 6;
x[1]*x[2] == 2;
x[1] == 1;
NonLinears.solve();
```

After this program fragment, each of elements 1 through 4 of array `x` contains the value of its index.

[Derman and Van Wyk 1982] call this algorithm the "slightly nonlinear equation solver," and note that it could be extended to a broader class of equations by recognizing the forms of other simple nonlinear equations. For example, `3/x == 1/y` could be translated to `3*y == x` by simple cross-multiplication. The present implementation does not provide any such equation templates to extend the solver's capabilities.

A C Implementation

The implementation of the equation solver described in [Derman and Van Wyk 1984] occupies several hundred lines of `yacc`, `lex`, and C code that construct expression trees, as well as a largish function that evaluates the trees during a postorder traversal. The body of the evaluation function is a several-hundred line `switch` on the type of the node; it is riddled with casts because the internal and external nodes of expression trees are stored in `structs` of different types. It is awkward to experiment with extensions to the solver because several places in the code must change to add new node types or equation templates.

At first, I undertook to rewrite the solver in C++ because it offered the promise of avoiding the huge `switch` and the proliferation of type casts, localizing the changes needed to extend the solver, and allowing nicer notation by overloading the arithmetic operators. It turned out also to allow some simplification of the data structures and associated invariants.

The solver described in [Derman and Van Wyk 1984] maintains two data-structure invariants:

1. Dependency representations are stored as *ordered linear combinations*: the terms of an ordered linear combination appear in descending order by serial number of the variable, with the constant term at the end.
2. Dependency representations contain only independent variables.

Invariant 1 defines a canonical representation for linear combinations that makes it possible to form the linear combination of two ordered linear combinations in time proportional to their combined length. A function to perform this operation lies at the heart of both the C and the C++ implementations of the solver: it is used to copy ordered linear combinations and to eliminate variables from them, as well as to form ordered linear combinations of ordered linear combinations. The difference between the implementations is in the data structures they use: the C implementation keeps ordered linear combinations in a completely separate data structure from the one used for expression trees, while the C++ implementation derives the data structures for ordered linear combinations from those for generic nodes in expression trees. Thus, the C++ implementation has simpler data structures and allows more code to be reused.

To maintain Invariant 2, the C implementation keeps a list of all dependent variables; when variable `x` becomes dependent, the solver traverses this list and updates any dependency representations that include `x`, then adds `x` to the list of dependent variables; to keep the list as short as possible, variables are removed from the list when they become known. Invariant 2 is natural when we think of dependency representations as data structures in which linear combinations are stored. When they are viewed as objects, however, the following simpler invariant suffices:

2'. The result of evaluating a variable may contain only independent variables.

To maintain this invariant in the C++ implementation, a variable evaluates itself by recursively evaluating any variables involved in its dependency representation, then combining the answers to form the most up-to-date possible answer.

The C++ Implementation

As suggested by the examples above, the user of the C++ version of the solver declares the variables of a problem to be of type `Expr`, then constrains them by writing equations. Among the member functions of an object of type `Expr` are those to tell whether it represents an expression that is known, linear, dependent, or numeric, and those with which to print or evaluate the expression.

An object of class `Expr` contains a pointer to an object of some type derived from `Node`, and all of the member functions of `Expr` merely transfer the responsibility to the `Node` to which the `Expr` points. Class `Expr` does include a variety of constructors so that the different kinds of expressions that arise in equations are recognized and converted into `Expr`s. Class `Expr` also keeps track of reference counts in `Node`s, so that when a `Node` is no longer pointed to by any expression, the `Node` is freed automatically.

Class `Node` is an abstract base class from which the various kinds of nodes that can compose expression trees are derived:

```
Node
        Null
        Real
        RealVar
        Sum
                LinComb
        Diff
        Product
                Term
        Quotient
```

`Node`s have no public functions or members whatsoever; the user's exclusive access to `Node`s is through `Expr`, which is a friend of all `Node`s.

The names of most `Node`s are meant to suggest their role in expression trees; here are a few comments on their implementation.

A `Real` contains a real value, and may also contain a string that it uses to print itself. For example, one might write:

```
Expr pi = 3.14159265; pi.nameset("pi");
```

so that the formula for circumference would print as `2*pi*r` rather than as `2*3.14159265*r`.

A `RealVar` stores (in its member `deprep`) the dependency representation for a real variable. The constructor for `RealVar` initializes the dependency representation for variable `x` to `1*x+0`, which betokens the variable's independence; it also assigns the variable a unique serial number for use in maintaining

Invariant 1. Eventually the == operator, which is a friend of the class, might change the dependency representation to reflect information the solver has deduced about the values of the variables. Here is how a RealVar evaluates itself:

```
Expr RealVar::eval()
{
        if (dependent())
                deprep = deprep.eval();
        return deprep;
}
```

Thus, if the RealVar is independent or known, its deprep is simply returned; otherwise, the evaluation and subsequent assignment of its deprep leaves it with only independent variables on its deprep. This strategy (lazy evaluation) is what permitted the abandonment of Invariant 2 mentioned above.

Each of Sum, Diff, Product, and Quotient has two Exprs as operands, left and right. Here is the evaluation function for Sum:

```
Expr Sum::eval()
{
        Expr l = left.eval();
        Expr r = right.eval();
        if (l.linear() && r.linear())
                return add(1.0, l, 1.0, r);
        return l + r;
}
```

(add(c1, L1, c2, L2) forms the ordered linear combination representing c1*L1 + c2*L2.) If its operands are both linear combinations, Sum can return a linear combination. If either operand is not a linear combination, Sum just returns the sum of the two operands; notice though, that the calls to their member functions eval() might have produced some simplification in left and right, and these simplified versions are the operands to the result of Sum::eval().

The evaluation function for Product is only slightly more complicated:

```
Expr Product::eval()
{
        Expr l = left.eval();
        Expr r = right.eval();
        if (l.numeric() && r.linear())
                return add(l.numval(), r, 1.0, NullExpr());
        if (l.linear() && r.numeric())
                return add(r.numval(), l, 1.0, NullExpr());
        return l * r;
}
```

A product can be represented as a linear combination whenever one operand is a number and the other is a linear combination. Otherwise, Product::eval() returns the product of its operands after they have themselves been simplified.

A Term is a Product whose left operand is known to be a Real and whose right operand is known to be a RealVar (or Null, if the Term is constant). A Term adopts the serial number of its right operand. A LinComb is a Sum whose left operand is known to be a Term and whose right operand is another LinComb whose left operand's serial number is smaller than that of the Term on the left. Thus, LinCombs define the data structure with which to preserve Invariant 1, and all evaluation functions return LinCombs whenever possible.

Operators +, binary and unary −, *, and / are overloaded so that when their operands are Exprs they return Exprs that point to Nodes of the appropriate type (Sum, Diff, Product, or Quotient).

Operator == is overloaded to perform the basic step of the solver algorithm when its operands are Exprs. When both operands are linear, == imitates partial pivoting by making dependent the variable whose coefficient is largest in absolute value. If either operand is not linear, == enqueues the equation on

EqnQueue NonLinears. To indicate the disposition of the equation, == returns one of three values in an enumerated type: INCONSISTENT, CONSISTENT, or NONLINEAR.

The queue of nonlinear equations belongs to class EqnQueue. This class includes only three member functions: the constructor, which creates an empty queue; enqueue(), which adds an equation to the queue; and solve(), which applies == to solve as many equations on the queue as it can.

Comments

Early in the implementation I decided to make Expr an immutable type. This prevents surprises caused by over-eager simplification: the user who has defined z to be x*x + y + 4 will not come back later to find that z is y + 5 because x was determined somehow to be -1. Thus, one can be sure that the evaluator will not alter Exprs that are to be used in other ways (for example one might also differentiate an Expr symbolically; cf. [Jerrell 1989]).

Sometimes, however, one is happy to have the original expressions disappear as they are simplified. Evidently the user who types

```
Expr z = x*x + y + 4;
x == 1;
z = z.eval();
```

does not care if the original quadratic form goes away. To make this work as automatically as possible, I adopted a reference-counting scheme on Nodes, inspired by Andy Koenig's work [Koenig 1988].

Since Exprs just contain pointers to Nodes, an Expr that has been passed to a function can access and alter members of the Node to which the Expr points. This is how function aux() in the first example can impose constraints on variables defined in main(). Because expr is counting the references to each Node, in fact, all functions *must* receive Expr arguments, and not Expr& arguments.

Thus, this approach brings some disadvantages as well. First, calls to constructors for Exprs are interposed whenever Exprs are passed as arguments to or returned by functions. When the solver processes a dense linear system on ten variables, for example, it calls malloc() almost 8,000 times. This represents a constant factor of only eight on the asymptotic running time of Gauss-Jordan elimination, which would not be worrisome but for the known worst-case properties of malloc().

The second disadvantage is stylistic: when new Node types that involve member functions not defined by the other Nodes are added to the library, dummy functions must still be added to the base classes Node and Expr. To illustrate, consider this declaration for class Node (Expr contains a similar collection of functions):

```
class Node {
        friend class Expr;
        int refct;
  protected:
        Node();
        virtual int serial();
        virtual boolean null();
        virtual void print();
        virtual boolean known();
        virtual boolean linear();
        virtual boolean dependent();
        virtual Expr eval();
        virtual boolean numeric();
        virtual number numval();
        virtual Expr varget();
        virtual void varset(Expr);
        virtual char *nameget();
        virtual void nameset(char *);
        virtual Expr rest();
        virtual ~Node();
};
```

Even though only the `Real` and `RealVar` `Nodes` can use `nameset()`, for example, a function `nameset()` must be defined for *all* `Nodes`. The same holds for many of the other member functions, including `serial()`, `numval()`, `varget()`, `varset()`, and `rest()`. Fortunately the base class can define these functions to cause program abortion, which ensures that they are not called from an inappropriate `Expr`. On the other hand, it seems I need to change the base class whenever I derive another class from it, which mitigates some of the advantages I sought using C++ in the first place.

Future Directions

The slightly nonlinear equation solver is certainly not top-of-the-line numerical analysis software. Among its advantages, however, are that it does not require that the system be linear in form, that its simple solution algorithm makes it possible to offer helpful diagnostics when a system of equations is inconsistent, and that it is not hard to extend it with simple template-matching rules such as transforming `sqrt(x) == c` to `x == c*c`, or `a/x == b/y` to `a*y == b*x`. One could also imagine using the slightly nonlinear solver as a preprocessor to remove as much linearity as possible from a system of equations before passing it on to a higher-powered solver for nonlinear equations.

Another possibility, of course, would be to eschew altogether the framework of the slightly nonlinear equation solver. For example, instead of processing linear equations as soon as they are encountered, as in the present implementation, one could merely add each equation to a set as it was asserted, then called a solving function to find and apply a minimum-degree or other elimination order. Much of the code for the C++ implementation could still be used in such an endeavor.

Acknowledgements

Thanks to Brian Kernighan, Rob Pike, and Howard Trickey for helpful comments.

References

Emanuel Derman and Christopher J. Van Wyk, ''A simple equation solver and its application to financial modeling,'' *Software—Practice and Experience* 14 (1984), 1169-1181.

Max E. Jerrell, ''Function minimization and automatic differentiation using C++,'' *Proc. OOPSLA '89* (1989), 169-173.

Andrew R. Koenig, ''An example of dynamic binding in C++,' *Journal of Object-Oriented Programming* 1 (1988), 60-62.

Christopher J. Van Wyk, ''A high-level language for specifying pictures,'' *ACM Transactions on Graphics* 1 (1982), 163-182.

LogiC++: An Integrated Logic and Object-Oriented Programming Language

Shaun-inn Wu
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota 56267
Electronic mail: shauninn@caa.mrs.umn.edu

ABSTRACT

The ability to structure and organize knowledge in problem domains is very important in knowledge representation. Object-oriented programming languages facilitate the modularization of the knowledge of interest as a class hierarchy. With class inheritance they have benefits for representing taxonomic knowledge. In addition, the feature of passing messages in object-oriented programming can easily represent the interactions among different components. Hence objected-oriented programming is well suited to be the framework of knowledge representation. However, it is also important to have the high expressive power of the representation language when designing knowledge-based systems. The means of representing methods in most object-oriented programming languages is too procedural and less declarative and expressive. Logic programming with declarative semantics can contribute to the expressiveness of representing methods. Thus it is desirable to take the advantage of both logic and object-oriented programming by combining them into one environment.

LogiC++ integrates logic and object-oriented programming. It is primarily based on object-oriented programming. C++ was chosen to provide the framework of object-oriented programming because of it's compatibility with C, its efficiency, and its compile-time checking capability. However, the logic programming language Prolog can be used to express methods for the objects. Prolog is a programming language used for many artificial intelligence applications including knowledge representation. Its syntax is very simple and its semantics is based on a simple yet powerful concept: the resolution principle. In this paper, we describe a compiler that takes as input a C++ program with methods written as Horn clauses in Prolog and that produces an equivalent C++ program as the output. The C++ program can then be compiled by a C++ compiler.

1. Introduction

Object-oriented programming provides a uniform notion of objects with encapsulated state and well-defined messages together with inheritance. Methods in most object-oriented programing languages are represented as procedures. Hence it is not easy to represent knowledge of declarative nature. On the other hand, logic programming based on first-order predicate calculus provides a declarative framework with unification and backtracking. One can construct a program by describing what is true in the problem domain instead of how to solve the problem. Thus it is natural to represent knowledge-based systems in logic. In addition, one can develop methods incrementally by adding newly discovered knowledge in the application domain without affecting existing knowledge. Hence it is very desirable to combine logic programming and object-oriented programming to get the advantages of both.

In general there are many approaches for combining logic programming and object-oriented programming. The first approach is to incorporate the ideas of object-oriented programming into logic programming. There are several alternatives for incorporating the notions of objects into logic programming. For example, objects can be characterized by intentions which are functions from states to values (Chen and Warren, 1988). A different scheme is to introduce in Prolog the syntactic notations specifying messages and inheritance (Zaniolo, 1984). The notation for expressing classes, objects and methods in Concurrent Prolog are provided by a preprocessor Vulcan (Kahn, et. al., 1986). Yet another way is to represent objects and messages by literals; procedures that modify objects are implemented by a different type of formula and invoked by a new inference rule that arguments SLD resolution (Conery, 1988).

The second approach to combining logic programming and object-oriented programming is to add logic programming capabilities into an object-oriented framework. In SPOOL, methods are represented as logic programs and messages to an object as goal invocations within that object (Fukunaga and Hirose, 1986). In Orient84/K, a definition of knowledge objects contains logic programs as their knowledge parts (Ishikawa and Tokoro, 1986).

Many other approaches are possible, too. For instance, an object-oriented programming language such as Loops can be interfaced with a logic programming language such as Quintus Prolog (Koschmann and Evens, 1988). In KSL/Logic, the notions of logic programming are simulated in object-oriented programming (Ibrahim and Cummins, 1990a, 1990b). More research is necessary to identify the advantages and disadvantages of different approaches but this is beyond the scope of this paper.

For our work, we basically take the second approach. We started with an object-oriented framework provided by C++ (Stroustrup, 1986). In LogiC++, a program is basically a C++ program. However, methods can be represented by Prolog programs (Clocksin and Mellish, 1984). There are many advantages for doing so. First of all, the advantages of C++ such as the efficiency and compile-time checking are kept in our integrated language. Second, the expressive power of Prolog based on declarative semantics is obtained for defining methods. Moreover, the language features in both languages are altered at a minimum level so that programmers can have a very smooth transition from both C++ and Prolog to LogiC++. In this paper, we describe a compiler that takes as input a C++ program with methods written as Prolog programs and produces an equivalent C++ program as the output. The C++ program can then be compiled and run on many systems of different configurations.

A Prolog compiler was previously designed and implemented to take a Prolog program as input and produces the object codes in C (Halverson, et. al., 1990; Wu, 1990). Our LogiC++ compiler incorporates this Prolog compiler as the major component. In this paper, the descriptions of LogiC++ compiler as well as this Prolog compiler is presented.

2. The Integrated Language: LogiC++

In general a program in LogiC++ has a very strong resemblance to C++ programs. A C++ program consists of definitions of classes, functions and data structures. A class contains the data structure and the operations that implement an abstract data type. The operations, usually called methods, can be performed on the underlying type. In a C++

program, these methods are usually implemented as functions. However, in a LogiC++ program the keyword *methods* is used to indicate that the following methods would be defined by Horn clauses as in Prolog except that scope qualifiers can be declared for predicates. The rest of the program structure is basically the same as any C++ program.

Prolog basically makes use of the clausal form of first-order predicate logic and allows, at most, one predicate in the head of a clause but any number in the body. If the body is empty, it is a *unit clause* or a *fact* which asserts some fact; if the head is missing, it is called a *goal clause* or a *query*; otherwise, it is called a *rule*. These clauses are usually referred to as *Horn clauses*. A Prolog program is constructed from a number of rules and unit clauses, and one goal clause.

<u>EXAMPLE</u> The following is a sample program in LogiC++:

```
class manager;

class employee {
    private:
        manager *his_manager;
    public:
        employee(void) { his_manager = 0; };
        employee(manager &m) { his_manager = &m; };
        methods
            request(Subject) :- his_manager.give_me_approval(Subject).
}

class manager : public employee {
    private:
        methods
            unimportant(domestic_trip).
            unimportant(presentation).
            unreasonable(double_salary).
    public:
        manager(void) : employee() { };
        manager(manager &m) : employee(m) { };
        methods
            give_me_approval(Subject) :- unimportant(Subject).
            give_me_approval(Subject) :- not unreasonable(Subject).
}

main() {
    . . .
    manager charlie;
    employee snoopy(charlie);          // charlie is snoopy's manager

    if (snoopy.request(domestic_trip)) {
        . . .
    }
        . . .
}
```

In the above program, two classes *employee* and *manager* are defined where *employee* is a super class of *manager*. One public method *request* is defined for *employee*. For the class *manager*, a public method *give_me_approval* and two private methods *unimportant* and *unreasonable* are defined as a local knowledge base. Then an object *charlie* of the class *manager* is instantiated with nobody as his manager and another object *snoopy* of the class *employee* is instantiated with the object *charlie* as his manager. When a message *request* with an argument *domestic_trip* is sent to *snoopy*, a message *give_me_approval* with that argument is being sent to his manage *charlie* to obtain his approval. The result of *request* depends on the result of *give_me_approval*. This program is compiled into an equivalent C++ program as shown in the Appendix. The resulted C++ program will then be compiled into the executable codes by a C++ compiler.

Assuming that we learn more information about what's unimportant and what's unreasonable, we could easily add the information into the knowledge base. For instance, if it's unreasonable for an employee to take two months off, we only need to add *unreasonable(take_two_months_off).* into the private section of the class *manager*. The rest of the program could stay unchanged and the new information will be naturally included in our knowledge base.

Moreover, assume that another class *upper_level_manager* is defined as a derived class of *manager*. We can add the method *important(international_trip).* into the private knowledge base of *manager* and add the method *give_me_approval(Subject) :- important(Subject), his_manager.give_me_approval(Subject).* into the public section of *manager*. The *upper_level_manager* class contains more knowledge and hence may grant the employees' requests passed by the lower-level managers. Hence a hierarchy of management can be set up in which each level of managers can have additional information in their knowledge base.

The beauty of integrating logic programming into object-oriented programming is that the knowledge can be increased gracefully. The original program doesn't need any change other than including additional knowledge expressed in logical clauses.

3. Method Compilation

Our methods are basically Horn clauses as in Prolog. Our compiler takes methods as input and produces C++ functions as output. In the compiler, the symbol table holds the names for head predicates. These C++ functions are then compiled by a C++ compiler and run on different systems.

Methods are compiled according to the procedural interpretation of Prolog program execution (which will be discussed in the Section 4). Each clause of a method is compiled into a function in C++. The C++ functions for all the clauses with the same head predicate are grouped into a single function in which each clause is represented as an alternative of a *switch* statement. Predicates with the same name and arity are considered the same predicate. In the body of clauses, subgoal calls are compiled into appropriate function calls. These function calls attempt to make each of the clauses with the unifiable head predicate succeed using the unification process.

Besides the codes generated for clauses and predicates, the C++ object code for methods also contains the codes manipulating all the necessary data structures at runtime. Much of the C++ code produced for each of the predicates simply manipulates data on these runtime data structures.

The actual compilation of methods is done in two phases. During the first phase of the compilation, the Horn clauses of methods are hashed into a symbol table which contains all the necessary information for the generation of the C++ code. During the second phase, the C++ codes are actually generated according to the information stored in the symbol table.

An entry in the symbol table contains the name and arity of a predicate and a list of the structures for the clauses with this predicate as their head predicate. In the structure for a clause, there are structures representing those predicates in the body of the clause and all the logical variables in the clause. In the structure for a predicate, the type of the predicate and its logical variables are included.

4. Method Execution

There are generally two interpretations of logic program execution. The procedural interpretation is discussed here. The inferential interpretation can be found elsewhere (e.g., Hogger, 1984). In the procedural interpretation of logic program execution, goals are viewed as sets of procedure calls. A clause is viewed as a procedure with its head predicate as the procedure name and formal parameters and the body of the clause as the procedure body in which predicates are a sequence of procedure calls. Each goal is processed by calling an appropriate procedure which is a clause whose head predicate is unifiable to the goal. The goal is then replaced by the procedure calls in the procedure body after unifying the actual and formal parameters.

For a subgoal call occurring in the body of a clause, the symbol table is searched first for a unifiable predicate; if there is one, it is used to continue the unification process. If no unifiable predicate is found, the subgoal fails and the unification process backtracks to the previous subgoal and tries to find another unifiable predicate.

Each procedure call produces a frame; that is, an activation record. The frame consists of the data needed by the call on the runtime data structures. This includes the data for logical variables and the runtime environment such as where to go if the call succeeds, fails, or backtracks. The running of an object program involves the unification of logical variables and the maintenance of control flow by using the values in the frame on those runtime data structures.

Our runtime model for methods is based on the two-stack representation developed for the DEC-10 Prolog (Warren, 1977). The runtime data structures include both the local and the global stack and the trail. The local stack contains the local variables not referred to by the variables in earlier calls and control variables to keep track of certain information such as the current procedure call, where to backtrack, ..., etc. The global stack, on the other hand, is used to keep track of those other variables referred to by the variables in earlier calls. The trail holds the variables which must be reset when backtracking occurs. The runtime environment consists of all the runtime data structures and the procedures manipulating these data structures which are needed in the execution of logical methods. The codes for the runtime environment is generated along with the compilation of Horn clauses of methods.

5. Discussion

The combination of logic and object-oriented programming shows a great potential to support the representation of domain knowledge into declarative data types and make them reusable in different contexts.

For our work, we started with an object-oriented framework provided by C++ where methods are represented by functions. LogiC++ keeps many features of C++ such as objects, classes, inheritance and the advantages of C++ such as the efficiency and compile-time checking. In addition, definition of methods can take advantage of the declarative semantics and expressive power of Prolog. Because most language features in both languages are maintained, C++ and Prolog programmers may have a smooth transition to programming in LogiC++.

Our system is designed primarily on the top of C++. Despite some problems existing in C++ (Sakkinen, 1988), the main point of this work is to show the power of combining the two powerful paradigms of logic programming and object-oriented programming. Our compiler behaves more like a pre-processor that translates a LogiC++ program into a C++ program. In the future we hope to incorporate this feature of expressing method by logic programs into a C++ compiler.

Although our work presents a relatively clean design that fits logical methods well with a C++ framework, it is just a first-stub attack to the general problem of fully unifying logic programming and object-oriented programming. The integration in our work seems seamless and smooth at a glance. However, a closer examination exposes the inconsistencies between the parameter passing, evaluation rules and uses of literals of C++ functions and logical methods. It would be extremely helpful to have a broader logic programming base so that any object, class, variables and literals of different types can be unified with logical variables.

This suggests that it would seem appropriate to redesign an object-oriented *Warren Abstract Machine* (Warren, 1983), possibly implemented in C++, as a first step towards fully integrating logic programming and object-oriented programming. However, much more work needs to be done on deeper questions such as the semantics of such integrated languages.

References

Chen, Weidong and David Scott Warren, *Objects as Intentions*, Proceedings of 1988 International Conference on Logic Programming, 1988, pp. 404 - 419.

Clocksin, W. F. and C. S. Mellish, Programming in Prolog, Springer-Verlag, Berlin, 1984.

Conery, John S., *Logical Objects*, Proceedings of 1988 International Conference on Logic Programming, 1988, pp. 420 - 434.

Fukunaga, Koichi and and Shin-ichi Hirose, *An experience with a Prolog-based Object-Oriented Language*, Proceedings of OOPSLA'86, Portland, Oregon, 1986, pp. 224 - 231.

Halverson, Tom, Dennis Van Dam and Shaun-inn Wu, Morlog Technical Manual, Computer Science Discipline, Division of Science and Mathematics, University of Minnesota, Morris, 1990.

Hogger, Christophor J., Introduction to Logic Programming, Academic Press, London, 1984.

Ibrahim, Mamdouh H. and Fred A. Cummins, *Objects with Logic*, Proceedings of ACM Computer Science Conference, Washington, D.C., 1990a, pp. 128 - 133.

Ibrahim, Mamdouh H. and Fred A. Cummins, *KSL/Logic: Integration of Logic with Objects*, Proceedings of IEEE International Conference on Computer Languages, New Orleans, Louisiana, 1990b, pp. 228 - 235.

Ishikawa, Yutaka and Mario Tokoro, *A Concurrent Object-oriented Knowledge Representation Language Orient84/K: Its Features and Implementation*, Proceedings of OOPSLA'86, Portland, Oregon, 1986, pp. 232-241.

Kahn, K., E. D. Tribble, M. S. Miller and D. G. Bobrow, *Objects in Concurrent Logic Programming Languages*, Proceedings of OOPSLA'86, Portland, Oregon, 1986, pp. 242 - 257.

Koschmann, T. and M. W. Evens, *Bridging the Gap Between Object-Oriented and Logic Programming*, IEEE Software, Volume 5, Number 5, July 1988, pp. 36 - 42.

Sakkinen, Markku, *On the Darker Side of C++*, Proceedings of European Conference on Object-Oriented Programming, Oslo, Norway, August 15-17, 1988, pp. 162-176.

Stroustrup, Bjarne, The C++ Programming Language, Addison-Wesley, Readings, Mass., 1986.

Warren, David H. D., Implementing Prolog - Compiling Predicate Logic Programs, Volume 1 and 2, DAI Research Report Number 39 and 40, Department of Artificial Intelligence, University of Edinburgh, 1977.

Warren, David H. D., An Abstract Prolog Instruction Set, SRI Technical Note 309, 1983.

Wu, Shaun-inn, *Crafting a Prolog Compiler: A Project in Compiler Design*, Proceedings of the 23rd Small College Computing Symposium, River Falls, Wisconsin, 1990, pp. 252 - 262.

Zaniolo, C., *Object-Oriented Programming in Prolog*, Proceedings of 1984 IEEE Symposium on Logic Programming, Atlantic City, 1984, pp. 265 - 270.

Appendix

The following is the C++ program produced by the LogiC++ compiler for the example program. Only a few translated functions are shown here due to the limitation on the length of this paper.

```
#include "rtinc.c"  // Routines manipulating run-time structures for logical methods

class manager;

class employee {
    private:
        manager *his_manager;
    public:
        employee(void) { his_manager = 0; };
        employee(manager &m) { his_manager = &m; };
        request1b0o1(int inparams[9], int localframe *ourparent);
        request1b0(int howfar, int inparams[9], int localframe *ourparent);
```

```
};

class manager : public employee {
    private:
        unimportant1b0o1(int inparams[9], int localframe *ourparent);
        unimportant1b0o2(int inparams[9], int localframe *ourparent);
        unimportant1b0(int howfar, int inparams[9], int localframe *ourparent);
        unreasonable1b0o1(int inparams[9], int localframe *ourparent);
        unreasonable1b0(int howfar, int inparams[9], int localframe *ourparent);
    public:
        manager(void) : employee() {};
        manager(manager &m) : employee(m) {};
        give_me_approval1b0o1(int inparams[9], int localframe *ourparent);
        give_me_approval1b0o2(int inparams[9], int localframe *ourparent);
        give_me_approval1b0(int howfar, int inparams[9], int localframe *ourparent);
};
. . .
manager::unreasonable1b0o1(int inparams[9], int localframe *ourparent) {
    struct localframe *thisparent; int i,j; struct vars *wvar,*wvar2; int didfail = succeed;
    int predfail[10]; static int params[9] = { 0, 1, 0, 0, 0, 0, 0, 0, 0 };
    if (backtracking == 0) {
        createenv(ourparent);
        thisparent = curlocal;
        addlocvar("double_salary",2,"double_salary");
        for(i=1;i <= 1;i++)
          {wvar=curlocal->firstvar;
           wvar2=ourparent->firstvar;
           for(j=1;j < params[i];j++)
           wvar=wvar->next;
           for(j=1;j < inparams[i];j++)
           wvar2=wvar2->next;
           if (unify(wvar2,wvar) == fail)
             didfail = fail;}
    }
    else if (backtracking == 1){
        if (curplace == thebottom) thisparent = curlocal->parent;
        else thisparent = curplace->parent;}
        if (didfail == succeed) {return(succeed);}
        else {return(fail);}
};

manager::unreasonable1b0(int howfar, int inparams[9], int localframe *ourparent) {
    int q; struct localframe *thisparent; int temphowfar;
    level++;
    while(1 == 1)
        {temphowfar = howfar;
        howfar = changehowfar(howfar);
        choice = howfar;
        switch(howfar) {
            case 1 : q = unreasonable1b0o1(inparams,ourparent);
                     if (q == succeed) {level = level-1;
                         return(howfar+1);}
                     else {howfar++;
                         ourparent =curlocal->parent;
```

```
                    poplocal();
                    backtracking = 0;
                    break;}
          default : level = level - 1;
                    backtracking = 1;
                    return(fail);}}};

main() {
    . . .
    manager  charlie;
    employee  snoopy(charlie);
    int tryit;  int j;  int k; int passparams[9];
    . . .
    createenv(curlocal);
    backloc = curlocal;
    thebottom = curlocal;
    curplace = thebottom;
    addlocvar("",2,"domestic_trip");
    for(j=0;j<=1;j++)
        passparams[j] = j;
    if (snoopy.request1b0(j,passparams,curlocal)) {
        . . .
    }
        . . .
}
```

THE USENIX ASSOCIATION

The USENIX Association is a not-for-profit organization of those interested in UNIX and UNIX-like systems. It is dedicated to fostering and communicating the development of research and technological information and ideas pertaining to advanced computing systems, to the monitoring and encouragement of continuing innovation in advanced computing environments, and to the provision of a forum where technical issues are aired and critical thought exercised so that its members can remain current and vital.

To these ends, the Association conducts large semi-annual technical conferences and sponsors workshops concerned with varied special-interest topics; publishes proceedings of those meetings; publishes a bimonthly newsletter *;login:*; produces a quarterly technical journal, *Computing Systems*; co-publishes books with The MIT Press; serves as coordinator of an exchange of software; and distributes 4.3BSD manuals and 2.10BSD tapes. The Association also actively participates in and reports on the activities of various ANSI, IEEE and ISO standards efforts.

Computing Systems, published quarterly in conjunction with the University of California Press, is a refereed scholarly journal devoted to chronicling the development of advanced computing systems. It uses an aggressive review cycle providing authors with the opportunity to publish new results quickly, usually within six months of submission.

The USENIX Association intends to continue these and other projects, and in addition, the Association will focus new energies on expanding the Association's activities in the areas of outreach to universities and students, improving the technical community's visibility and stature in the computing world, and continuing to improve its conferences and workshops.

The Association was formed in 1975 and incorporated in 1980 to meet the needs of the UNIX users and system maintainers who convened periodically to discuss problems and exchange ideas concerning UNIX. It is governed by a Board of Directors elected biennially.

There are four classes of membership in the Association, differentiated primarily by the fees paid and services provided.

For further information about membership or to order publications, contact:

USENIX Association
Suite 215
2560 Ninth Street
Berkeley, CA 94710
Telephone: 415/528-8649
Email: office@usenix.org
Fax: 415/548-5738

USENIX SUPPORTING MEMBERS

Aerospace Corporation
AT&T Information Systems
Digital Equipment Corporation
Frame Technology, Inc.
Quality Micro Systems
Matsushita Graphic Communication Systems, Inc.
mt Xinu
Open Software Foundation
Sun Microsystems, Inc.
Sybase, Inc.
UUNET Technologies, Inc.